ТЕХНОЛОГИИ ПОСТРОЕНИЯ

WEB-ИНТЕРФЕЙСОВ

ЧАСТЬ 2

ПРЕДИСЛОВИЕ

Основные методические рекомендации по освоении дисциплины «Технологии построения web-интерфейсов» обучающимся даны в предисловии к учебно-методическому пособию «Технологии построения web-интерфейсов. Часть 1».

Вторая часть учебно-методического пособия состоит из трех глав. По основным темам и разделам дисциплины обучающимся важно придерживаться следующих методических рекомендаций.

В первой главе «Язык программирования JavaScript» обучающимся необходимо усвоить основы языка JavaScript, типы данных, функции, операторы и другие важные конструкции данного языка, особенности работы с ним в браузерах, а также возможности и особенности применения библиотеки jQuery.

Во второй главе «Стандарт HTML 5» рассказывается оо основных возможностях пятого стандарта языка разметки гипертекста и его отличиях от других версий HTML, виды и новые возможности API-интерфейсов.

В третьей главе «Адаптивная верстка» рассматриваются основы и наиболее важные особенности адаптивной верстки сайтов, инструментарий и виды текстовых редакторов, используемых при создании web-конструкций. В этой теме обучающимся необходимо также уяснить особенности применения библиотеки Bootstrap при верстке.

В конце каждой главы приводится список вопросов и заданий для контроля уровня освоения дисциплины обучающимися.

Глава 1. Язык программирования JavaScript

## **Введение в JavaScript**

JavaScript изначально создавался для того, чтобы сделать web-странички «живыми». Программы на этом языке называются скриптами. В браузере они подключаются напрямую к HTML и, как только загружается страничка – тут же выполняются.

### **Возможности JavaScript**

В браузере с помощью JavaScript можно вносить изменения в структурустраницы, взаимодействовать с посетителями и сервером:

* создавать новые HTML-теги, удалять существующие, менять стили элементов, прятать, показывать элементы;
* реагировать на действия посетителя, обрабатывать клики мыши, перемещения курсора, нажатия на клавиатуру;
* посылать запросы на сервер и загружать данные без перезагрузки страницы (эта технология называется "AJAX");
* получать и устанавливать cookie, запрашивать данные, выводить сообщения.

JavaScript – быстрый и мощный язык, но браузер накладывает на его исполнение некоторые ограничения. Это сделано для безопасности пользователей, чтобы злоумышленник не мог с помощью JavaScript получить личные данные или как-то навредить компьютеру пользователя. Этих ограничений нет там, где JavaScript используется вне браузера, например, на сервере. Кроме того, современные браузеры предоставляют свои механизмы по установке плагинов и расширений, которые обладают расширенными возможностями, но требуют специальных действий по установке от пользователя

Большинство возможностей JavaScript в браузере ограничено текущим окном и страницей:

1. JavaScript не может читать/записывать произвольные файлы на жесткий диск, копировать их или вызывать программы. Он не имеет прямого доступа к операционной системе.
2. Современные браузеры могут работать с файлами, но эта возможность ограничена специально выделенной директорией. Возможности по доступу к устройствам также прорабатываются в современных в современных стандартах и частично доступны в некоторых браузерах.
3. JavaScript, работающий в одной вкладке, не может общаться с другими вкладками и окнами, за исключением случая, когда он сам открыл это окно или несколько вкладок из одного источника (одинаковый домен, порт, протокол).
4. Из JavaScript можно легко посылать запросы на сервер, с которого пришла страница. Запрос на другой домен тоже возможен, но менее удобен, т. к. и здесь есть ограничения безопасности.

Синтаксис JavaScript устраивает не всех: одним он кажется слишком свободным, другим – наоборот, слишком ограниченным, третьи хотят добавить в язык дополнительные возможности, которых нет в стандарте. Это нормально, ведь требования и проекты у всех разные.

В последние годы появилось много языков, которые добавляют различные возможности «поверх» JavaScript, а для запуска в браузере – при помощи специальных инструментов «трансляторов» превращаются в обычный JavaScript-код. Это преобразование происходит автоматически и совершенно прозрачно, при этом неудобств в разработке и отладке практически нет.

При этом разные языки выглядят по-разному и добавляют совершенно разные вещи:

* например, язык CoffeeScript сосредоточен на большей ясности и краткости кода. Как правило, его особенно любят программисты на Ruby;
* язык Dart интересен тем, что он не только транслируется в JavaScript, как и другие языки, но и имеет свою независимую среду выполнения, которая даёт ему ряд возможностей и доступна для встраивания в приложения (вне браузера). Он разрабатывается компанией Google.

## **Основы языка JavaScript**

**В чем писать код?**

Notepad++ − свободный текстовый редактор с открытым исходным кодом для Windows с подсветкой синтаксиса большого количества языков программирования и разметки.

IDE (Integrated Development Environment) – «интегрированная среда разработки», означает редактор, который расширен большим количеством «наворотов», умеет работать со вспомогательными системами, такими как багтрекер, контроль версий, и много чего ещё. Как правило, IDE загружает весь проект целиком, поэтому может предоставлять автодополнение по функциям всего проекта, удобную навигацию по его файлам и т.п.

Браузеры − клавиша F12 под Windows или комбинация клавиш Cmd+Opt+J под Mac откроет инструменты разработчика и вкладку Console, в которой можно править код. Можно набирать и запускать команды, область для команд на рисунке находится справа, запуск команд осуществляется нажатием Ctrl+Enter (для Mac – Cmd+Enter).

Программы на JavaScript могут быть вставлены в любое место HTML-документа с помощью тега <script>. Для примера:

<!DOCTYPE HTML>

<html>

<body>

<p>Перед скриптом...</p>

<script>

alert( 'Привет, мир!' );

</script>

<p>...После скрипта.</p>

</body>

</html>

Можно запустить пример, нажав на кнопку «Play» в правом верхнем углу блока с кодом выше. Тег <script> содержит JavaScript-код, который автоматически выполнится, когда браузер его обработает.

[**Современная разметка**](file:///C:\Users\user\Desktop\Web-интерфейсы\Часть%202%20материалы\1.2%20Привет,%20мир!.html#sovremennaya-razmetka)

Тег <script> имеет несколько атрибутов, которые редко используются, но всё ещё могут встретиться в старом коде:

Атрибут type: <script type=…>

Старый стандарт HTML, HTML4, требовал наличия этого атрибута в теге <script>. Обычно он имел значение type="text/JavaScript". На текущий момент этого больше не требуется. Более того, в современном стандарте HTML смысл этого атрибута полностью изменился. Теперь он может использоваться для JavaScript-модулей.

Атрибут language: <script language=…>

Этот атрибут должен был задавать язык, на котором написан скрипт. Но так как JavaScript является языком по умолчанию, в этом атрибуте уже нет необходимости.

**[Внешние скрипты](file:///C:\\Users\\user\\Desktop\\Web-интерфейсы\\Часть%202%20материалы\\1.2%20Привет,%20мир!.html" \l "vneshnie-skripty)**

Если у вас много JavaScript-кода, вы можете поместить его в отдельный файл. Файл скрипта можно подключить к HTML с помощью атрибута src:

<script src="/path/to/script.js"></script>

Здесь /path/to/script.js – это абсолютный путь до скрипта от корня сайта. Также можно указать относительный путь от текущей страницы. Например, src="script.js" будет означать, что файл "script.js" находится в текущей папке.

Можно указать и полный URL-адрес. Например:

<script src="https://cloudflare.com/Ajax/libs/lodash.js"> </script>

Для подключения нескольких скриптов используйте несколько тегов:

<script src="/js/script1.js"></script>

<script src="/js/script2.js"></script>

Как правило, только простейшие скрипты помещаются непосредственно в HTML-код. Более сложные выделяются в отдельные файлы. Польза от отдельных файлов в том, что браузер загрузит скрипт отдельно и сможет хранить его в [кеше](https://en.wikipedia.org/wiki/Web_cache). Другие страницы, которые подключают тот же скрипт, смогут брать его из кеша вместо повторной загрузки из сети. И таким образом файл будет загружаться с сервера только один раз. Это сокращает расход трафика и ускоряет загрузку страниц.

Если атрибут src установлен, содержимое тега script будет игнорироваться.

В одном теге <script> нельзя использовать одновременно атрибут src и код внутри. Нижеприведённый пример не работает:

<script src="file.js">

alert(1); // содержимое игнорируется, так как есть

атрибут src

</script>

Нужно выбрать: либо внешний скрипт <script src="…">, либо обычный код внутри тега <script>. Вышеприведённый пример можно разделить на два скрипта:

<script src="file.js"></script>

<script>

alert(1);

</script>

В отличие от, например, PHP точка с запятой коду JavaScript, имеющему в строке одну инструкцию, не требуется. Поэтому следующая строка вполне имеет право на существование:

x += 10

Но при необходимости иметь в строке более одной инструкции их нужно разделить точками с запятыми:

x += 10; y -= 5; z = 0

Последнюю точку с запятой можно опустить, поскольку последняя инструкция будет завершена символом новой строки. В правилах использования точки с запятой есть исключения. Если пишутся URL-закладки, JavaScript или инструкция завершается ссылкой на переменную или функцию и первый символ расположенной ниже строки является левой круглой или фигурной скобкой, нужно обязательно поставить точку с запятой, иначе сценарий JavaScript даст сбой. Поэтому при любых сомнениях нужно ставить точку с запятой.

В силу общих наследственных черт, приобретенных у языка программирования C, JavaScript имеют много общего и между собой, в частности в приемах комментирования кода. В первую очередь это касается однострочных комментариев:

// Это комментарий

В этой технологии используется пара прямых слешей (//), информирующая JavaScript о том, что все остальные символы должны быть проигнорированы. А затем наступает черед многострочных комментариев:

/\* Это раздел многострочного комментария, не подвергаемого интерпретации \*/

Многострочный комментарий начинается с последовательности символов /\* и заканчивается символами \*/. Нужно лишь запомнить, что использовать вложенные многострочные комментарии не допускается, поэтому важно убедиться в отсутствии большого закомментированного участка кода, в котором уже имеются многострочные комментарии.

В JavaScript нет никаких идентификационных символов переменных. Вместо этого в отношении имен переменных действуют следующие правила:

* имена переменных могут включать только буквы a–z, A–Z, цифры 0–9, символ $ и символ подчеркивания (\_);
* никакие другие символы, включая пробелы или знаки пунктуации, использовать в именах переменных не допускается'
* первым в имени переменной может быть символ из диапазонов a–z, A–Z, символ $ или символ подчеркивания \_ (и никаких цифр)'
* имена чувствительны к регистру. Имена Count, count и COUNT принадлежат трем разным переменным;
* ограничений на длину имени переменной не существует.

JavaScript допускает использование символа $, и он может быть первым символом в имени переменной или функции.

### Типы данных

В JavaScript существует несколько основных типов данных:

**Number** − единый тип число используется как для целых, так и для дробных чисел. Существуют специальные числовые значения Infinity (бесконечность) и NaN (ошибка вычислений). Например, бесконечность Infinity получается при делении на ноль.

**String** − в некоторых языках программирования есть специальный тип данных для одного символа. Например, в языке С это char. В JavaScript есть только тип «строка» string (в JavaScript одинарные и двойные кавычки равноправны).

**Boolean** − у него всего два значения: true (истина) и false (ложь). Как правило, такой тип используется для хранения значения типа да/нет.

**Null** − значение null не относится ни к одному из типов выше, а образует свой отдельный тип, состоящий из единственного значения null. В JavaScript null не является «ссылкой на несуществующий объект» или «нулевым указателем», как в некоторых других языках. Это просто специальное значение, которое имеет смысл «ничего» или «значение неизвестно».

**Undefined** − значение undefined, как и null, образует свой собственный тип, состоящий из одного этого значения. Оно имеет смысл «значение не присвоено». Если переменная объявлена, но в неё ничего не записано, то её значение как раз и есть undefined. Можно присвоить undefined и в явном виде, хотя это делается редко.

**Object** − используется для коллекций данных и для объявления более сложных сущностей. Объявляются объекты при помощи фигурных скобок {…}.

В JavaScript нет явного преобразования типов, осуществляемого с помощью операторов (int) или (float). Когда нужно, чтобы значение имело определенный тип данных, используется одна из функций преобразования типа JavaScript, показанных в таблице 1.1.

**Таблица 1.1.** Типы данных и функции JavaScript.

|  |  |
| --- | --- |
| Преобразование в тип данных | Используемая функция |
| Int, Integer | parseInt() |
| Bool, Boolean | Boolean() |
| Float, Double, Real | parseFloat() |
| String | String() |
| Array | split() |

### Строки

В JavaScript любые текстовые данные являются строками.

Получение длины символами происходит следующим образом:

var str = "My\n"; // 3 символа.

alert( str.length ); // 3

Получение символа на позиции, при первой позиции равной 0:

var str = "jQuery";

alert( str.charAt(0) ); // "j"

Или же…

var str = "Я - современный браузер!";

alert( str[0] ); // "Я"

str[0]

Содержимое строки в JavaScript нельзя изменять. Нельзя взять символ посередине и заменить его. Как только строка создана – она такая навсегда. Можно лишь создать целиком новую строку и присвоить в переменную вместо старой, например:

var str = "строка";

str = str[3] + str[4] + str[5];

alert(str); // ока

str.indexOf

Для поиска подстроки есть метод **str.indexOf** (подстрока[начальная позиия]). Он возвращает позицию, на которой находится подстрока или -1, если ничего не найдено. Например:

var str = "Widget with id";

alert( str.indexOf("Widget") ); // 0, т.к. "Widget" найден прямо в начале str

alert( str.indexOf("id") ); // 1, т.к. "id" найден, начиная с позиции 1

alert( str.indexOf("widget") ); // -1, не найдено, так как поиск учитывает регистр

Метод substring(start, end) возвращает подстроку с позиции start до, но не включая end:

var str = "stringify";

alert(str.substring(0,1)); // "s", символы с позиции 0 по 1 не включая 1.

### Операторы в JavaScript

Операторы в JavaScript могут использоваться в математических операциях, для внесения изменений в строки, а также в операциях сравнения и логических операциях (И, ИЛИ и т. д.). Математические операторы JavaScript представлены в таблице 1.2.

**Таблица 1.2.** Арифметические операторы .

|  |  |
| --- | --- |
| Оператор | Описание |
| + | Сложение |
| - | Вычитание |
| \* | Умножение |
| / | Деление |
| % | Деление по модулю (остаток от деления) |
| ++ | Инкремент |
| –– | Декремент |

Операторы присваивания используются для присваивания значений переменным (таблица 1.3).

**Таблица 1.3.** Операторы присваивания

|  |  |
| --- | --- |
| Оператор | Пример |
| = | a=9 |
| += | a=c+2 |
| += | a=c+"string" |
| -= | a=c-5 |
| \*= | a=c\*3 |
| /= | a=c/3 |
| %= | a=c%7 |

Операторы сравнения обычно используются с такими конструкциями, как инструкция if, в которой требуется сравнивать два элемента (таблица 1.4).

**Таблица 1.4.** Операторы сравнения.

|  |  |  |
| --- | --- | --- |
| Оператор | Описание | Пример |
| == | Равно | a== |
| != | Не равно | a!= |
| > | Больше | a> |
| < | Меньше | a< |
| >= | Больше или равно | a>== |
| <= | Меньше или равно | a<== |
| === | Равно (и того же типа) | a=== |
| !== | Не равно (и того же типа) | a!== |

В Javascrypt также активно применяются логические переменные (таблица 1.5):

**Таблица 1.5**. Логические переменные.

|  |  |  |
| --- | --- | --- |
| Оператор | Описание | Пример |
| && | И | j == 1 && k == 2 |
| || | Или | j < 100 || j > 0 |
| ! | Не | ! (j == k) |

### Взаимодействие с пользователем

Функция **alert** выводит на экран окно с сообщением и приостанавливает выполнение скрипта, пока пользователь не нажмёт «ОК»

Окно сообщения, которое выводится, является модальным окном. Слово «модальное» означает, что посетитель не может взаимодействовать со страницей, нажимать другие кнопки и т.п., пока не завершит работу с окном. В данном случае – пока не нажмёт на «OK».

alert( "Привет" );

Функция **prompt** выводит модальное окно с заголовком title, полем для ввода текста, заполненным строкой по умолчанию default и кнопками OK/CANCEL. Пользователь должен либо что-то ввести и нажать OK, либо отменить ввод кликом на CANCEL или нажатием Esc на клавиатуре.

result = prompt(title, default);

Функция **confirm** выводит окно с вопросом question с двумя кнопками: OK и CANCEL. Результатом будет true при нажатии OK и false – при CANCEL(Esc).

var isAdmin = confirm("Вы - администратор?")

alert( isAdmin );

### Условия

Условия изменяют процесс выполнения программы. Они позволяют задавать конкретные вопросы и по-разному реагировать на полученные ответы. Существуют три типа условий, не связанных с циклами: инструкция if, инструкция switch и оператор ?.

**Инструкция if**

Код внутри этой инструкции выполняется только в том случае, если заданное выражение вычисляется как true. Многострочные инструкции if заключаются в фигурные скобки, но для однострочных инструкций скобки можно опустить. Поэтому допустимы все следующие инструкции:

if (a > 100) {

b=2

document.write("a больше 100")

}

if (b == 10)

document.write("b равно 10")

**Инструкция else**

Если условие не было соблюдено, то с помощью инструкции else может быть выполнен альтернативный блок кода:

if (a > 100) {

document.write("a больше 100")

}

else {

document.write("a меньше или равно 100")

}

В JavaScript, нет инструкции elseif, но ее отсутствие компенсируется возможностью использования инструкции else, за которой следует еще одна инструкция if, чем создается эквивалент инструкции elseif:

if (a > 100) {

document.write("a больше 100")

}

else if(a < 100) {

document.write("a меньше 100")

}

else {

document.write("a равно 100")

}

За инструкцией else после новой инструкции if точно так же может следовать еще одна инструкция if и т. д.

Несмотря на то что в этих инструкциях использованы фигурные скобки, наличие внутри каждой пары этих скобок всего одной строки кода позволяет переписать весь пример следующим образом:

if (a > 100) document.write("a больше 100")

else if(a < 100) document.write("a меньше 100")

else document.write("a равно 100")

**Инструкция switch**

Инструкция switch применяется в том случае, когда одиночная переменная или результат вычисления выражения может иметь несколько значений, для каждого из которых нужно применить свою функцию.

Предположим, пользователю доступны следующие варианты: Home, About, News, Login, Links − и переменной page присваивается одно из этих значений, соответствующее тому, что ввел пользователь.

Код, созданный для этого с помощью конструкции if...else if..., может иметь вид, показанный в примере 1.1.

**Пример 1.1.** Многострочная конструкция if...else if

<script>

if (page == "Home") document.write("Вы выбрали Home")

else if (page == "About") document.write("Вы выбрали About")

else if (page == "News") document.write("Вы выбрали News")

else if (page == "Login") document.write("Вы выбрали Login")

else if (page == "Links") document.write("Вы выбрали Links")

</script>

А при использовании конструкции switch код может иметь вид, показанный в примере 1.2.

**Пример 1.2.** Конструкция switch.

<script>

switch (page) {

case "Home":

document.write("Вы выбрали Home")

break

case "About":

document.write("Вы выбрали About")

break

case "News":

document.write("Вы выбрали News")

break

case "Login":

document.write("Вы выбрали Login")

break

case "Links":

document.write("Вы выбрали Links")

break

}

</script>

Переменная page здесь присутствует только в самом начале инструкции switch. После чего совпадения проверяются командой case. При совпадении выполняется условная инструкция. Команда break позволяет сценарию прекратить работу инструкции switch при соблюдении условия. Если вы не хотите, чтобы выполнение всех инструкций, начиная со следующей case, продолжилось, не забудьте поставить команду break.

С помощью ключевого слова default для инструкции switch можно определить действие по умолчанию на тот случай, когда не будет выполнено ни одно из условий. В примере 1.3 показан фрагмент кода, который может быть вставлен в код примера 1.2.

**Пример 1.3.** Инструкция default.

default:

document.write("Нераспознанный выбор")

break

**Оператор ?**

Тернарный оператор, состоящий из вопросительного знака (?), применяемого в сочетании с символом двоеточия (:), является упрощенной формой текста if...else. В примере 1.4 показано использование тернарного оператора.

**Пример 1.4.** Использование тернарного оператора.

<script>

document.write(

a <= 5

? "a меньше или равно 5"

: "a больше 5"

)

</script>

### Циклы

В JavaScript поддерживаются циклы while, do...while и for.

**Циклы while**

В JavaScript в циклах while сначала проверяется значение выражения, а выполнение инструкций внутри цикла начинается лишь в том случае, если выражение вычисляется как true. Если выражение вычисляется как false, управление переходит к следующей инструкции JavaScript (если таковая имеется). После завершения итерации цикла выражение опять проверяется на истинность и процесс продолжается до тех пор, пока не наступит момент, когда выражение будет вычислено как false, или пока выполнение не будет остановлено по какой-нибудь другой причине. Этот цикл показан в примере 1.5.

**Пример 1.5.** Цикл while.

<script>

counter=0

while (counter < 5)

{

document.write("Счетчик: " + counter + "<br>")

++counter

}

</script>

Этот сценарий выведет следующую информацию:

Счетчик: 0

Счетчик: 1

Счетчик: 2

Счетчик: 3

Счетчик: 4

Если бы переменная counter не увеличивалась на единицу внутри цикла, то вполне возможно, что некоторые браузеры перестали бы откликаться из-за входа в бесконечный цикл и работу со страницей было бы трудно остановить даже нажатием клавиши Esc или кнопки остановки загрузки страницы. Поэтому к циклам в JavaScript нужно относиться с большой осторожностью.

**Циклы do...while**

Когда нужен цикл, в котором еще до того, как будет проверено выражение, должна пройти хотя бы одна итерация, используется цикл do...while, который похож на цикл while, за исключением того, что проверка выражения осуществляется только после каждой итерации цикла. Поэтому для вывода первых семи результатов таблицы умножения на 7 можно воспользоваться кодом, показанным в примере 1.6.

**Пример 1.6.** Цикл do...while.

<script>

count = 1

do {

document.write(count + " умножить на 7 равно " + count \* 7 + "<br>")

}

while (++count <= 7)

</script>

Этот цикл выведет следующую информацию:

1 умножить на 7 равно 7

2 умножить на 7 равно 14

3 умножить на 7 равно 21

4 умножить на 7 равно 28

5 умножить на 7 равно 35

6 умножить на 7 равно 42

7 умножить на 7 равно 49

**Циклы for**

Цикл for принимает три параметра:

* выражение инициализации;
* выражение условия;
* выражение модификации.

Эти параметры отделяются друг от друга точками с запятыми: for (выражение1; выражение2; выражение3). В примере 1.7 показан код такого цикла.

**Пример 1.7.** Использование цикла for

<script>

for (count = 1 ; count <= 7 ; ++count) {

document.write(count + " умножить на 7 равно " + count \* 7 + "<br>");

}

</script>

В первом параметре цикла for можно присваивать значения сразу нескольким переменным, разделяя выражения запятыми:

for (i = 1, j = 1; i < 10; i++)

Точно так же в последнем параметре можно осуществлять сразу несколько модификаций:

for (i = 1; i < 10; i++, --j)

Или можно одновременно делать и то и другое:

for (i = 1, j = 1; i < 10; i++, --j)

**Прекращение работы цикла**

Команду break можно применять и внутри циклов for. Использование команды break показано в примере 1.8.

**Пример 1.8.** Использование команды break в цикле for.

<script>

haystack = new Array()

haystack[17] = "Иголка"

for (j = 0 ; j < 20 ; ++j) {

if (haystack[j] == "Иголка") {

document.write("<br>- найдена в элементе " + j)

break

}

else document.write(j + ", ")

}

</script>

Этот сценарий выводит следующую информацию:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,

- найдена в элементе 17

**Инструкция continue**

Иногда нужно не выйти из цикла, а пропустить выполнение тех инструкций, которые остались в данной итерации. В таких случаях можно воспользоваться командой continue. Ее применение показано в примере 1.9.

**Пример 1.9.** Использование команды continue в цикле for.

<script>

haystack = new Array()

haystack[4] = "Иголка"

haystack[11] = "Иголка"

haystack[17] = "Иголка"

for (j = 0 ; j < 20 ; ++j) {

if (haystack[j] == "Иголка") {

document.write("<br>- найдена в элементе " + j + "<br>")

continue

}

document.write(j + ", ")

}

</scr ipt>

Обратите внимание на то, что второй вызов метода document.write не нужно помещать в инструкцию else (как было в предыдущем примере), поскольку, если будет найдено совпадение, то в результате выполнения команды continue данный вызов будет пропущен. Этот сценарий выводит следующую информацию:

0, 1, 2, 3,

- найдена в элементе 4

5, 6, 7, 8, 9, 10,

- найдена в элементе 11

12, 13, 14, 15, 16,

- найдена в элементе 17

18, 19,

### Функции

В JavaScript наряду с доступом к десяткам встроенных функций (или методов), среди которых и метод write, который, как мы уже видели, использовался в вызовах document.write, можно создавать и собственные функции. Как только появляется какой-нибудь непростой фрагмент кода с перспективами на многократное использование, он становится кандидатом на оформление в виде функции.

В общем виде синтаксис функции выглядит следующим образом:

function имя\_функции([параметр[, ...]])

{

инструкции

}

Из первой строки синтаксиса следует, что:

* определение начинается со слова function;
* следующее за этим словом имя должно начинаться с буквы или символа подчеркивания, за которым следует любое количество букв, цифр, символов доллара или подчеркивания;
* необходимо использовать скобки.
* дополнительно могут применяться один или несколько параметров, разделенных запятыми (о чем свидетельствуют квадратные скобки, не являющиеся частью синтаксиса функции).

Имена функций чувствительны к регистру букв, поэтому строки getInput, GETINPUT и getinput ссылаются на разные функции.

В JavaScript для имен функций действует общепринятое соглашение: первая буква каждого слова в имени, за исключением первой буквы всего имени, должна быть прописной. Поэтому в приведенных примерах имен предпочтение следует отдать имени getInput, имеющему формат, используемый большинством программистов. Это соглашение часто называют bumpyCaps (неровностями из прописных букв) или camelCase (как горбы у верблюда).

Инструкции, которые будут выполняться после вызова функции, начинаются с открывающей фигурной скобки, а составляющая ей пару закрывающая фигурная скобка должна завершать перечень этих инструкций. Среди инструкций обязаны присутствовать одна или несколько инструкций **return**, которые заставляют функцию прекратить выполнение и вернуть управление вызвавшему ее коду. Если к инструкции **return** прилагается какое-нибудь значение, то вызывающий код может его извлечь.

### Массивы

**Числовые массивы**

Чтобы создать новый массив, нужно воспользоваться следующим синтаксисом:

Arrayname = new Array()

Или же его более краткой формой:

arrayname = []

Для присваивания значений элементам массива в JavaScript используют метод **push:**

arrayname.push("Элемент1")

arrayname.push("Элемент2")

Он позволяет добавлять к массиву элементы, не отслеживая их количество. Когда потребуется узнать, сколько элементов содержится в массиве, можно будет воспользоваться свойством **length**:

document.write(arrayname.length)

Если нужно будет проконтролировать размещение элементов, расставляя их по конкретным местам, можно воспользоваться другим синтаксисом:

arrayname[0] = "Элемент 1"

arrayname[1] = "Элемент 2"

В примере 1.10 показан простой сценарий, в котором создается массив, в него загружается несколько элементов, после чего эти элементы отображаются на экране.

**Пример 1.10.** Создание, построение и вывод массива на экран.

<script>

numbers = []

numbers.push("Один")

numbers.push("Два")

numbers.push("Три")

for(j = 0 : j<numbers.length : ++j)

document.write("Элемент" + j +"="+ numbers[j]+"<br>")

</script>

Этот сценарий выводит следующую информацию:

Элемент 0 = Один

Элемент 1 = Два

Элемент 2 = Три

После этого ничто не помешает добавить к данному массиву дополнительные элементы. Теперь в вашем распоряжении есть несколько способов добавления элементов к массиву и один способ ссылки на них, но JavaScript предлагает куда более обширный арсенал способов. Но сначала рассмотрим еще один тип массива.

**Ассоциативные массивы**

К **ассоциативным** относятся такие массивы, в которых ссылки на элементы осуществляются по именам, а не по числовому смещению. Чтобы создать ассоциативный массив, нужно определить блок элементов, заключенный в фигурные скобки. Для каждого элемента слева от двоеточия (:) указывается его ключ, а справа − содержимое. В примере 1.11 показано, как можно создать ассоциативный массив для хранения данных о товаре в разделе интернет – магазина канцелярии.

**Пример 1.11.** Создание и отображение ассоциативного массива.

<script>

balls = {"синий": "синие карандаши, 8",

"красный": "красные карандаши, 6",

"зеленый": "зеленые карандаши, 4",

"желтый": "желтые карандаши, 2 шт."}

for(ball in balls)

document.write(ball +"="+ balls[ball]+"<br>")

</script>

Для проверки факта создания и заполнения массива я воспользовался еще одной разновидностью цикла for, в которой применяется ключевое слово in. В этом цикле создается новая переменная, которая задействуется только внутри массива (в данном примере − ball), и вызывается последовательный перебор всех элементов массива, указанных справа от ключевого слова in (в данном примере − balls). Цикл обрабатывает каждый элемент массива balls, помещая значение ключа в переменную ball.

Используя значение ключа, сохраненное в переменной ball, можно также получить значение текущего элемента массива balls. Результат вызова сценария этого примера в браузере будет иметь следующий вид:

синий = синие карандаши, 8

красный = красные карандаши, 6

зеленый = зеленые карандаши, 4

желтый = желтые карандаши, 2 шт.

Чтобы получить значение конкретного элемента ассоциативного массива, нужно в явном виде указать его ключ (в данном случае будет выведено значение зеленые карандаши, 4):

document.write(balls['зеленые'])

**Многомерные массивы**

В JavaScript для создания многомерного массива нужно просто поместить массивы внутрь других массивов. Например, чтобы создать массив, содержащий сведения о двумерной шахматной доске (8×8 клеток), можно воспользоваться кодом примера 1.12.

**Пример 1.12.** Создание многомерного числового массива.

<script>

checkerboard = Array(

Array(' ', 'o', ' ', 'o', ' ', 'o', ' ', 'o'),

Array('o', ' ', 'o', ' ', 'o', ' ', 'o', ' '),

Array(' ', 'o', ' ', 'o', ' ', 'o', ' ', 'o'),

Array(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '),

Array(' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '),

Array('O', ' ', 'O', ' ', 'O', ' ', 'O', ' '),

Array(' ', 'O', ' ', 'O', ' ', 'O', ' ', 'O'),

Array('O', ' ', 'O', ' ', 'O', ' ', 'O', ' '))

document.write("<pre>")

for (j = 0 ; j < 8 ; ++j)

{ for (k = 0 ; k < 8 ; ++k)

document.write(checkerboard[j][k] + " ")

document.write("<br>")

}

document.write("</pre>")

</script>

В данном примере буквами нижнего регистра обозначены черные, а буквами верхнего регистра − белые фигуры. Два цикла for, один из которых является вложенным, осуществляют последовательный перебор элементов массива и отображают его содержимое. Внешний цикл содержит две инструкции, поэтому они заключены в фигурные скобки. Внутренний цикл обрабатывает каждую клетку в горизонтали, выводя символ, находящийся в позиции [j][k], за которым следует пробел (чтобы придать выводимой информации квадратную конфигурацию). В этом цикле содержится всего одна инструкция, поэтому заключать ее в фигурные скобки не имеет смысла. Теги <pre> и </pre> обеспечивают корректный вывод информации:

o o o o

o o o o

o o o o

O O O O

O O O O

O O O O

Можно также получить непосредственный доступ к любому элементу данного массива, применив для этого квадратные скобки:

document.write(checkerboard[7][2])

Эта инструкция выводит букву O верхнего регистра, то есть содержимое восьмой сверху и третьей справа клетки − индексация элементов в массиве начинается с нуля, а не с единицы.

**Методы массивов**

Реализовать возможности, предоставленные массивами, помогают имеющиеся в JavaScript готовые к использованию методы для работы с ними и с содержащимися в них данными. Рассмотрим подборку, состоящую из наиболее востребованных методов.

**Метод concat**

Метод concat объединяет два массива или ряд значений в массив. Например, следующий код выведет Банан, Виноград, Морковь, Капуста:

fruit = ["Банан", "Виноград"]

veg = ["Морковь", "Капуста"]

document.write(fruit.concat(veg))

В качестве аргументов можно указать несколько массивов, тогда метод concat добавит все их элементы в порядке указания массивов.

А вот еще один способ использования метода concat, где с массивом pets объединяются простые значения и на экран выводится строка Кошка, Собака,Рыба,Кролик,Хомяк:

pets = ["Кошка", "Собака", "Рыба"]

more\_pets = pets.concat("Кролик", "Хомяк")

document.write(more\_pets)

**Метод forEach (для браузеров не из семейства IE)**

Воспользоваться этим методом можно, передав ему имя функции, которая будет вызвана для каждого элемента массива. Как это делается, показано в примере 1.13.

**Пример 1.13.** Использование метода forEach.

<script>

pets = ["Кошка", "Собака", "Кролик", "Хомяк"]

pets.forEach(output)

function output(element, index, array)

{ document.write("Элемент с индексом " + index + "

содержит значение " + element + "<br>")

}

</script>

В данном случае функция, передаваемая методу forEach, называется output. Она воспринимает три параметра: элемент, его индекс и массив. Как они используются, зависит от потребностей вашей функции. В данном примере они просто отображают значения индекса и элемента с помощью метода document.write. После того как массив будет заполнен, можно вызвать рассматриваемый метод:

pets.forEach(output)

На выходе будет получена следующая информация:

Элемент с индексом 0 содержит значение Кошка

Элемент с индексом 1 содержит значение Собака

Элемент с индексом 2 содержит значение Кролик

Элемент с индексом 3 содержит значение Хомяк

**Метод join**

Метод join позволяет превратить все значения массива в строки, а затем объединить их в одну большую строку, расставляя между значениями необязательные разделители. В примере 1.14 показаны три способа использования этого метода.

**Пример 1.14.** Использование метода join.

<script>

pets = ["Кошка", "Собака", "Кролик", "Хомяк"]

document.write(pets.join() + "<br>")

document.write(pets.join(' ') + "<br>")

document.write(pets.join(' : ') + "<br>")

</script>

Если не указывать параметр, метод join использует в качестве разделителя элементов запятую, в противном случае между элементами вставляется переданная методу join строка. Код примера выводит следующую информацию:

Кошка,Собака,Кролик,Хомяк

Кошка Собака Кролик Хомяк

Кошка : Собака : Кролик : Хомяк

**Методы push и pop**

Применение метода push для вставки значения в массив уже было рассмотрено. Противоположным ему по действию является метод pop. Он удаляет последний вставленный элемент из массива и возвращает значение этого элемента. Порядок его использования показан в примере 1.15.

**Пример: 1.15.** Использование методов push и pop внутри.  
 цикла и за его пределами

<script>

sports = ["Футбол", "Теннис", "Бейсбол"]

document.write("Изначально = " + sports + "<br>") **sports.push("Hockey")**

document.write("После вставки = " + sports + "<br>")

removed = sports.pop()

document.write("После удаления = " + sports + "<br>") document.write("Удаленный элемент = " + removed + <br>")

</script>

Три основные инструкции этого сценария выделены полужирным шрифтом. Сначала в сценарии создается массив по имени sports, содержащий три элемента, затем в него вставляется четвертый элемент, после чего сценарий удаляет этот элемент. В процессе этих действий с помощью метода document.write отображаются разные значения массива. Сценарий выводит следующую информацию:

Изначально = Футбол,Теннис,Бейсбол

После вставки = Футбол,Теннис,Бейсбол,Хоккей

После удаления = Футбол,Теннис,Бейсбол

Удаленный элемент = Хоккей

Как показано в примере 1.16, методы push и pop применяются в тех случаях, когда нужно отвлечься от каких-нибудь действий на другие, а затем вернуться к прежним действиям.

**Пример 1.16.** Использование методов push и pop внутри цикла и за его пределами

<script>

numbers = []

for (j=0 ; j<3 ; ++j)

{

numbers.push(j); document.write("Вставлен элемент " + j + "<br>")

}

// Здесь осуществляются какие-нибудь другие действия document.write("<br>")

document.write("Удален элемент " + numbers.pop() + "<br>")

document.write("Удален элемент " + numbers.pop() + "<br>")

document.write("Удален элемент " + numbers.pop() + "<br>")

</script>

Код этого примера выведет следующую информацию:

Вставлен элемент 0

Вставлен элемент 1

Вставлен элемент 2

Удален элемент 2

Удален элемент 1

Удален элемент 0

**Метод reverse**

Использование метода reverse осуществляет простую перестановку элементов массива в обратном порядке, его действие показано в примере 1.17.

**Пример. 1.17.** Применение метода reverse.

<script>

sports = ["Футбол", "Теннис", "Бейсбол", "Хоккей"]

sports.reverse()

document.write(sports)

</script>

Исходный массив подвергается изменению, и сценарий выводит следующую информацию:

Хоккей, Бейсбол, Теннис, Футбол

**Метод sort**

Метод **sort** позволяет расставить все элементы массива в алфавитном или в каком-нибудь другом порядке в зависимости от применяемых параметров. В примере 1.18 показаны четыре типа сортировки.

**Пример 1.18.** Использование метода sort .

<script>

// Сортировка по алфавиту

sports = ["Футбол", "Теннис", "Бейсбол", "Хоккей"]

sports.sort() document.write(sports + "<br>")

// Сортировка по алфавиту в обратном порядке

sports = ["Футбол", "Теннис", "Бейсбол", "Хоккей"] sports.sort().reverse()

document.write(sports + "<br>")

// Сортировка чисел по возрастанию

numbers = [7, 23, 6, 74]

numbers.sort(function(a,b){return a - b})

document.write(numbers + "<br>")

// Сортировка чисел по убыванию

numbers = [7, 23, 6, 74]

numbers.sort(function(a,b){return b - a})

document.write(numbers + "<br>")

</script>

В первом из четырех блоков этого примера применяется сортировка по алфавиту, во втором − возвращение к исходному виду, а затем метод reverse, чтобы получить сортировку по алфавиту в обратном порядке.

Третий и четвертый блоки усложнены использованием функции для сравнения взаимоотношений между a и b. У нее отсутствует имя, поскольку она используется только при сортировке. Функция по имени function, которая применяется для создания безымянных функций, уже встречалась при определении метода класса (метода showUser).

Здесь function создает безымянную функцию, отвечающую запросам метода sort. Если функция возвращает значение больше нуля, сортировка предполагает, что b ставится перед a. Если функция возвращает значение меньше нуля, сортировка предполагает, что a ставится перед b. Сортировка запускает эту функцию применительно ко всем значениям массива для определения порядка их следования.

За счет манипуляции возвращаемыми значениями (a - b или b - a) в третьем и четвертом блоках примера, разобранного чуть выше, осуществляется выбор между сортировкой чисел по возрастанию и по убыванию

### Объекты

Объекты используются для хранения коллекций различных значений и более сложных сущностей. В JavaScript объекты используются очень часто, это одна из основ языка.

Объект может быть создан с помощью фигурных скобок {…} с необязательным списком свойств. Свойство – это пара «ключ: значение», где ключ – это строка (также называемая «именем свойства»), а значение может быть чем угодно.

Мы можем представить объект в виде ящика с подписанными папками. Каждый элемент данных хранится в своей папке, на которой написан ключ. По ключу папку легко найти, удалить или добавить в неё что-либо. Пустой объект («пустой ящик») можно создать, используя один из двух вариантов синтаксиса:

let user = new Object(); // синтаксис "конструктор объекта"

let user = {}; // синтаксис "литерал объекта"

Обычно используют вариант с фигурными скобками {...}. Такое объявление называют литералом объекта или литеральной нотацией.

**[Литералы и свойства](file:///C:\\Users\\user\\Desktop\\Web-интерфейсы\\Часть%202%20материалы\\1.2%20Объекты%20как%20массивы.html" \l "literaly-i-svoystva)**

При использовании литерального синтаксиса {...} мы сразу можем поместить в объект несколько свойств в виде пар «ключ: значение»:

let user = { // объект

name: "John", // под ключом "name" хранится значение "John"

age: 30 // под ключом "age" хранится значение 30

};

Свойства объекта также иногда называют полями объекта. У каждого свойства есть ключ (также называемый «имя» или «идентификатор»). После имени свойства следует двоеточие ":", и затем указывается значение свойства. Если в объекте несколько свойств, то они перечисляются через запятую.

В объекте user сейчас находятся два свойства:

1. Первое свойство с именем "name" и значением "John".
2. Второе свойство с именем "age" и значением 30.

Можно сказать, что наш объект user – это ящик с двумя папками, подписанными «name» и «age». Мы можем в любой момент добавить в него новые папки, удалить папки или прочитать содержимое любой папки.

Для обращения к свойствам используется запись «через точку»:

// получаем свойства объекта:

alert( user.name ); // John

alert( user.age ); // 30

Значение может быть любого типа. Давайте добавим свойство с логическим значением:

user.isAdmin = true;

Для удаления свойства мы можем использовать оператор delete:

delete user.age;

Имя свойства может состоять из нескольких слов, но тогда оно должно быть заключено в кавычки:

let user = {

name: "John",

age: 30,

"likes birds": true // имя свойства из нескольких слов должно быть в кавычках

};

Последнее свойство объекта может заканчиваться запятой:

let user = {

name: "John",

age: 30,

}

Это называется «висячая запятая». Такой подход упрощает добавление, удаление и перемещение свойств, так как все строки объекта становятся одинаковыми.

**[Квадратные скобки](file:///C:\\Users\\user\\Desktop\\Web-интерфейсы\\Часть%202%20материалы\\1.2%20Объекты%20как%20массивы.html" \l "kvadratnye-skobki)**

Для свойств, имена которых состоят из нескольких слов, доступ к значению «через точку» не работает:

// это вызовет синтаксическую ошибку

user.likes birds = true

Так происходит, потому что точка требует, чтобы ключ был именован по правилам именования переменных. То есть не имел пробелов, не начинался с цифры и не содержал специальные символы, кроме & и \_.

Для таких случаев существует альтернативный способ доступа к свойствам через квадратные скобки. Такой способ сработает с любым именем свойства:

let user = {};

// присваивание значения свойству

user["likes birds"] = true;

// получение значения свойства

alert(user["likes birds"]); // true

// удаление свойства

delete user["likes birds"];

В этом случае код написан корректно. Обратите внимание, что строка в квадратных скобках закавычена (подойдёт любой тип кавычек). Квадратные скобки также позволяют обратиться к свойству, имя которого может быть результатом выражения. Например, имя свойства может храниться в переменной:

let key = "likes birds";

// то же самое, что и user["likes birds"] = true;

user[key] = true;

Здесь переменная key может быть вычислена во время выполнения кода или зависеть от пользовательского ввода. После этого мы используем её для доступа к свойству. Это даёт нам большую гибкость. Пример:

let user = {

name: "John",

age: 30

};

let key = prompt("Что вы хотите узнать о пользователе?", "name");

// доступ к свойству через переменную

alert( user[key] ); // John (если ввели "name")

Запись «через точку» такого не позволяет:

let user = {

name: "John",

age: 30

};

let key = "name";

alert( user.key ); // undefined

**[Вычисляемые свойства](file:///C:\\Users\\user\\Desktop\\Web-интерфейсы\\Часть%202%20материалы\\1.2%20Объекты%20как%20массивы.html" \l "vychislyaemye-svoystva)**

Мы можем использовать квадратные скобки в литеральной нотации для создания вычисляемого свойства. Пример:

let fruit = prompt("Какой фрукт купить?", "Apple");

let bag = {

[fruit]: 5, // имя свойства будет взято из переменной fruit

};

alert( bag.Apple ); // 5, если fruit="Apple"

Смысл вычисляемого свойства прост: запись [fruit] означает, что имя свойства необходимо взять из переменной fruit. И если посетитель введёт слово "Apple", то в объекте bag теперь будет лежать свойство {Apple: 5}.

По сути, пример выше работает так же, как и следующий пример:

let fruit = prompt("Какой фрукт купить?", "Apple");

let bag = {};

// имя свойства будет взято из переменной fruit

bag[fruit] = 5;

Но первый пример выглядит лаконичнее. Мы можем использовать и более сложные выражения в квадратных скобках:

let fruit = 'Apple';

let bag = {

[fruit + 'Computers']: 5 // bag.AppleComputers = 5

};

Квадратные скобки дают намного больше возможностей, чем запись через точку. Они позволяют использовать любые имена свойств и переменные, хотя и требуют более громоздких конструкций кода.

Подведём итог: в большинстве случаев, когда имена свойств известны и просты, используется запись через точку. Если же нужно что-то более сложное, то мы используем квадратные скобки. Зарезервированные слова разрешено использовать как имена свойств.

Имя переменной не может совпадать с зарезервированными словами, такими как «for», «let», «return» и т.д. Но для свойств объекта такого ограничения нет:

let obj = {

for: 1,

let: 2,

return: 3

};

alert( obj.for + obj.let + obj.return ); // 6

В принципе, разрешены любые имена свойств, но есть специальное свойство \_\_proto\_\_, которое по историческим причинам имеет особое поведение. Например, его значение всегда должно быть объектом, и мы не можем установить для него никакое другое значение:

let obj = {};

obj.\_\_proto\_\_ = 5;

alert(obj.\_\_proto\_\_); // [object Object], работает не так, как мы ожидали

Как мы видим, присвоение примитивного значения 5 игнорируется.

Это может стать источником ошибок и даже уязвимостей, если мы намереваемся хранить в объекте произвольные пары «ключ:значение» и позволяем посетителям указывать ключи: посетитель может указать \_\_proto\_\_ в качестве ключа, и логика присваивания будет нарушена (как показано выше). Есть способ заставить объекты обрабатывать \_\_proto\_\_ как обычное свойство.

**Свойство из переменной**

В коде часто нам необходимо использовать существующие переменные как значения для свойств с тем же именем. Например:

function makeUser(name, age) {

return {

name: name,

age: age

// ...другие свойства

};

}

let user = makeUser("John", 30);

alert(user.name); // John

В примере выше название свойств name и age совпадают с названиями переменных, которые мы подставляем в качестве значений этих свойств. Такой подход настолько распространён, что существуют специальные короткие свойства для упрощения этой записи. Вместо name:name можно написать просто name:

function makeUser(name, age) {

return {

name, // то же самое, что и name: name

age // то же самое, что и age: age

// ...

};

}

Мы можем использовать как обычные свойства, так и короткие в одном и том же объекте:

let user = {

name, // тоже самое, что и name:name

age: 30

};

**Проверка существования свойства**

Особенность объектов в том, что можно получить доступ к любому свойству. Даже если свойства не существует – ошибки не будет! При обращении к свойству, которого нет, возвращается undefined. Это позволяет просто проверить существование свойства – сравнением его с undefined:

let user = {};

alert( user.noSuchProperty === undefined ); // true означает "свойства нет"

Также существует специальный оператор "in" для проверки существования свойства в объекте. Синтаксис оператора:

"key" in object

Пример:

let user = { name: "John", age: 30 };

alert( "age" in user ); // true, user.age существует

alert( "no\_age" in user ); // false, user. no\_age не существует

Обратите внимание, что слева от оператора in должно быть имя свойства. Обычно это строка в кавычках. Если мы опускаем кавычки, это значит, что мы указываем переменную, в которой находится имя свойства. Например:

let user = { age: 30 };

let key = "age";

alert( key in user ); // true, имя свойства было взято из переменной key

Обычно строгого сравнения "=== undefined" достаточно для проверки наличия свойства. Но есть особый случай, когда оно не подходит, и нужно использовать "in". Это когда свойство существует, но содержит значение undefined:

let obj = {

test: undefined

};

alert( obj.test ); // выведет undefined, значит свойство не существует?

alert( "test" in obj ); // true, свойство существует!

В примере выше свойство obj.test технически существует в объекте. Оператор in сработал правильно. Подобные ситуации случаются очень редко, так как undefined обычно явно не присваивается. Для «неизвестных» или «пустых» свойств мы используем значение null. Таким образом, оператор in является экзотическим гостем в коде.

**[Цикл «for…in»](file:///C:\\Users\\user\\Desktop\\Web-интерфейсы\\Часть%202%20материалы\\1.2%20Объекты%20как%20массивы.html" \l "tsikl-for-in)**

Для перебора всех свойств объекта используется цикл for..in. Этот цикл отличается от изученного ранее цикла for(;;). Синтаксис:

for (key in object) {

// тело цикла выполняется для каждого свойства объекта

}

К примеру, давайте выведем все свойства объекта user:

let user = {

name: "John",

age: 30,

isAdmin: true

};

for (let key in user) {

// ключи

alert( key ); // name, age, isAdmin

// значения ключей

alert( user[key] ); // John, 30, true

}

Обратите внимание, что все конструкции «for» позволяют нам объявлять переменную внутри цикла, как, например, let key здесь. Кроме того, мы могли бы использовать другое имя переменной. Например, часто используется вариант "for (let prop in obj)".

**[Упорядочение свойств объекта](file:///C:\\Users\\user\\Desktop\\Web-интерфейсы\\Часть%202%20материалы\\1.2%20Объекты%20как%20массивы.html" \l "uporyadochenie-svoystv-obekta)**

Упорядочены ли свойства объекта? Другими словами, если мы будем в цикле перебирать все свойства объекта, получим ли мы их в том же порядке, в котором мы их добавляли? Можем ли мы на это рассчитывать?

Ответ: свойства упорядочены особым образом: свойства с целочисленными ключами сортируются по возрастанию, остальные располагаются в порядке создания. Разберёмся подробнее. В качестве примера рассмотрим объект с телефонными кодами:

let codes = {

"49": "Германия",

"41": "Швейцария",

"44": "Великобритания",

// ..,

"1": "США"

};

for (let code in codes) {

alert(code); // 1, 41, 44, 49

}

Если мы делаем сайт для немецкой аудитории, то, вероятно, мы хотим, чтобы код 49 был первым. Но если мы запустим код, мы увидим совершенно другую картину:

* США (1) идёт первым
* затем Швейцария (41) и так далее.

Телефонные коды идут в порядке возрастания, потому что они обладают целочисленными свойствами, т.е. являются целыми числами: 1, 41, 44, 49. Термин «целочисленное свойство» означает строку, которая может быть преобразована в целое число и обратно без изменений.

То есть, "49" – это целочисленное имя свойства, потому что если его преобразовать в целое число, а затем обратно в строку, то оно не изменится. А вот свойства "+49" или "1.2" таковыми не являются:

// Math.trunc - встроенная функция, которая удаляет десятичную часть

alert( String(Math.trunc(Number("49"))) ); // "49", то же самое ⇒ свойство целочисленное

alert( String(Math.trunc(Number("+49"))) ); // "49", не то же самое, что "+49" ⇒ свойство не целочисленное

alert( String(Math.trunc(Number("1.2"))) ); // "1", не то же самое, что "1.2" ⇒ свойство не целочисленное

С другой стороны, если ключи не целочисленные, то они перебираются в порядке создания, например:

let user = {

name: "John",

surname: "Smith"

};

user.age = 25; // добавим ещё одно свойство

// не целочисленные свойства перечислены в порядке создания

for (let prop in user) {

alert( prop ); // name, surname, age

}

Таким образом, чтобы решить нашу проблему с телефонными кодами, мы можем схитрить, сделав коды не целочисленными свойствами. Добавления знака "+" перед каждым кодом будет достаточно. Пример:

let codes = {

"+49": "Германия",

"+41": "Швейцария",

"+44": "Великобритания",

// ..,

"+1": "США"

};

for (let code in codes) {

alert( +code ); // 49, 41, 44, 1

}

Теперь код работает так, как мы задумывали.

**[Копирование по ссылке](file:///C:\\Users\\user\\Desktop\\Web-интерфейсы\\Часть%202%20материалы\\1.2%20Объекты%20как%20массивы.html" \l "kopirovanie-po-ssylke)**

Одним из фундаментальных отличий объектов от простых типов данных является то, что они хранятся и копируются «по ссылке». Простые типы: строки, числа, логические значения – присваиваются и копируются «по значению».

Например:

let message = "Hello!";

let phrase = message;

В результате мы имеем две независимые переменные, каждая из которых хранит строку "Hello!".

Объекты ведут себя иначе: переменная хранит не сам объект, а его «адрес в памяти», другими словами «ссылку» на него. Пример:

let user = {

name: "John"

};

Сам объект хранится где-то в памяти. А в переменной user лежит «ссылка» на эту область памяти. Когда переменная объекта копируется – копируется ссылка, сам же объект не дублируется. Если мы представляем объект как ящик, то переменная – это ключ к нему. Копирование переменной дублирует ключ, но не сам ящик. Например:

let user = { name: "John" };

let admin = user; // копируется ссылка

Теперь у нас есть две переменные, каждая из которых содержит ссылку на один и тот же объект. Мы можем использовать любую из переменных для доступа к ящику и изменения его содержимого:

let user = { name: 'John' };

let admin = user;

admin.name = 'Pete'; // изменено по ссылке из переменной "admin"

alert(user.name); // 'Pete', изменения видны по ссылке из переменной "user"

Приведённый выше пример демонстрирует, что объект только один. Как если бы у нас был один ящик с двумя ключами и мы использовали один из них (admin), чтобы войти в него и что-то изменить, а затем, открыв ящик другим ключом (user), мы бы увидели эти изменения.

**[Сравнение объектов](file:///C:\\Users\\user\\Desktop\\Web-интерфейсы\\Часть%202%20материалы\\1.2%20Объекты%20как%20массивы.html" \l "sravnenie-obektov)**

Операторы равенства == и строгого равенства === для объектов работают одинаково.

Два объекта равны только в том случае, если это один и тот же объект.

Например, две переменные ссылаются на один и тот же объект, они равны:

let a = {};

let b = a; // копирование по ссылке

alert( a == b ); // true, обе переменные ссылаются на один и тот же объект

alert( a === b ); // true

В примере ниже два разных объекта не равны, хотя и оба пусты:

let a = {};

let b = {}; // два независимых объекта

alert( a == b ); // false

Для сравнений типа obj1 > obj2 или для сравнения с примитивом obj == 5 объекты преобразуются в примитивы.

**[Объекты-константы](file:///C:\\Users\\user\\Desktop\\Web-интерфейсы\\Часть%202%20материалы\\1.2%20Объекты%20как%20массивы.html" \l "obekty-konstanty)**

Объект, объявленный через const, может быть изменён, например:

const user = {

name: "John"

};

user.age = 25; // (\*)

alert(user.age); // 25

Может показаться, что строка (\*) должна вызвать ошибку, но нет, здесь всё в порядке. Дело в том, что объявление const защищает от изменений только само значение user. А в нашем случае значение user – это ссылка на объект, и это значение мы не меняем. В строке (\*) мы действуем внутри объекта, мы не переназначаем user.

Если же мы попытаемся присвоить user другое значение, то const выдаст ошибку:

const user = {

name: "John"

};

// Ошибка (нельзя переопределять константу user)

user = {

name: "Pete"

};

**[Клонирование и объединение объектов, Object.assign](file:///C:\\Users\\user\\Desktop\\Web-интерфейсы\\Часть%202%20материалы\\1.2%20Объекты%20как%20массивы.html" \l "klonirovanie-i-obedinenie-obektov-object-assign)**

Как мы узнали ранее, при копировании переменной объекта создаётся ещё одна ссылка на тот же самый объект. Но что, если нам всё же нужно дублировать объект? Создать независимую копию, клон? Это выполнимо, но немного сложно, так как в JavaScript нет встроенного метода для этого. На самом деле, такая нужда возникает редко. В большинстве случаев нам достаточно копирования по ссылке.

Но если мы действительно этого хотим, то нам нужно создавать новый объект и повторять структуру дублируемого объекта, перебирая его свойства и копируя их. Например так:

let user = {

name: "John",

age: 30

};

let clone = {}; // новый пустой объект

// скопируем все свойства user в него

for (let key in user) {

clone[key] = user[key];

}

// теперь в переменной clone находится абсолютно независимый клон объекта.

clone.name = "Pete"; // изменим в нём данные

alert( user.name ); // в оригинальном объекте значение свойства `name` осталось прежним – John.

Кроме того, для этих целей можно использовать метод [Object.assign](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/assign).

Синтаксис:

Object.assign(dest, [src1, src2, src3...])

Аргументы dest, и src1, ..., srcN (может быть столько, сколько нужно) являются объектами.

Метод копирует свойства всех объектов src1, ..., srcN в объект dest. То есть, свойства всех перечисленных объектов, начиная со второго, копируются в первый объект. После копирования метод возвращает объект dest.

Например, объединим несколько объектов в один:

let user = { name: "John" };

let permissions1 = { canView: true };

let permissions2 = { canEdit: true };

// копируем все свойства из permissions1 и permissions2 в user

Object.assign(user, permissions1, permissions2);

// now user = { name: "John", canView: true, canEdit: true }

Если принимающий объект (user) уже имеет свойство с таким именем, оно будет перезаписано:

let user = { name: "John" };

// свойство name перезапишется, свойство isAdmin добавится

Object.assign(user, { name: "Pete", isAdmin: true });

// now user = { name: "Pete", isAdmin: true }

Мы также можем использовать Object.assign для простого клонирования:

let user = {

name: "John",

age: 30

};

let clone = Object.assign({}, user);

Все свойства объекта user будут скопированы в пустой объект, и ссылка на этот объект будет в переменной clone. На самом деле, такое клонирование работает так же, как и через цикл, но короче.

До сих пор мы предполагали, что все свойства пользователя примитивны. Но свойства могут быть ссылками на другие объекты. Что с ними делать?

Например, есть объект:

let user = {

name: "John",

sizes: {

height: 182,

width: 50

}

};

alert( user.sizes.height ); // 182

Теперь при клонировании недостаточно просто скопировать clone.sizes = user.sizes, поскольку user.sizes – это объект, он будет скопирован по ссылке. А значит объекты clone и user в своих свойствах sizes будут ссылаться на один и тот же объект:

let user = {

name: "John",

sizes: {

height: 182,

width: 50

}

};

let clone = Object.assign({}, user);

alert( user.sizes === clone.sizes ); // true, один и тот же объект

// user и clone обращаются к одному sizes

user.sizes.width++; // меняем свойство в одном объекте

alert(clone.sizes.width); // 51, видим результат в другом объекте

Чтобы исправить это, мы должны в цикле клонирования делать проверку, не является ли значение user[key] объектом, и если это так – копируем и его структуру тоже. Это называется «глубокое клонирование».

Существует стандартный алгоритм глубокого клонирования, [Structured cloning algorithm](http://w3c.github.io/html/infrastructure.html#safe-passing-of-structured-data). Он решает описанную выше задачу, а также более сложные задачи. Чтобы не изобретать велосипед, мы можем использовать реализацию этого алгоритма из JavaScript-библиотеки [lodash](https://lodash.com/), метод [\_.cloneDeep(obj)](https://lodash.com/docs#cloneDeep).

## **JavaScript в браузере**

### Объекты браузера window

Сам по себе язык JavaScript не предусматривает работы с браузером. Он вообще не знает про HTML. Но позволяет легко расширять себя новыми функциями и объектами.

На рис. 1.1 ниже схематически отображена структура, которая получается если посмотреть на совокупность браузерных объектов, доступных из JavaScript.

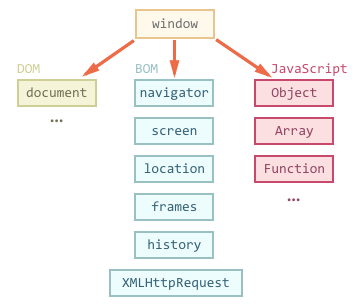


Рис.1.1. Структура объектов в окне браузера.

Как видно из рисунка, на вершине стоит window.

У этого объекта двоякая позиция – он с одной стороны является глобальным объектом в JavaScript, с другой – содержит свойства и методы для управления окном браузера, открытия новых окон, например:

// открыть новое окно/вкладку с URL http://ya.ru

window.open('http://ya.ru');

[**Объектная модель документа (DOM)**](https://learn.javascript.ru/browser-environment#объектная-модель-документа-dom) – глобальный объект document даёт возможность взаимодействовать с содержимым страницы. Он и громадное количество его свойств и методов описаны в [стандарте W3C DOM](http://www.w3.org/DOM/DOMTR). По историческим причинам когда-то появилась первая версия стандарта DOM Level 1, затем придумали ещё свойства и методы, и появился DOM Level 2, на текущий момент поверх них добавили ещё DOM Level 3 и готовится DOM 4.

[**Объектная модель браузера (BOM)**](https://learn.javascript.ru/browser-environment#объектная-модель-браузера-bom) – это объекты для работы с чем угодно, кроме документа.

Объект navigator содержит общую информацию о браузере и операционной системе. Особенно примечательны два свойства: navigator.userAgent – содержит информацию о браузере и navigator.platform – содержит информацию о платформе, позволяет различать Windows/Linux/Mac и т.п.

Объект location содержит информацию о текущем URL страницы и позволяет перенаправить посетителя на новый URL.

Функции alert/confirm/prompt – тоже входят в BOM.

Пример использования:

alert( location.href ); // выведет текущий адрес

В объекте **frames** содержатся window-объекты дочерних фреймов. Следующий код переводит фрейм на новый URL:

<iframe name="example" src="http://example.com" width="200" height="100"></iframe>

<script>

window.frames.example.location = 'http://example.com';

</script>

Объект **history** позволяет менять URL без перезагрузки страницы (в пределах того же домена) при помощи History API, а также перенаправлять посетителя назад-вперед по истории. Онне предоставляет возможности читать историю посещений. Можно отправить посетителя назад вызовом history.back() или вперед вызовом history.forward(), но сами адреса браузер не дает из соображений безопасности.

**DOM для взаимодействия с содержимым страницы**

Пример использования:

document.body.style.background = 'red';

alert('Элемент BODY стал красным, а сейчас обратно вернётся');

document.body.style.background = '';

## Дерево DOM

Рассмотрим пример использования формирования дерева DOM (рисунок 1.2)

<!DOCTYPE HTML>

<html>

<head>

<title>Заголовок</title>

</head>

<body>

…

</body>

</html>

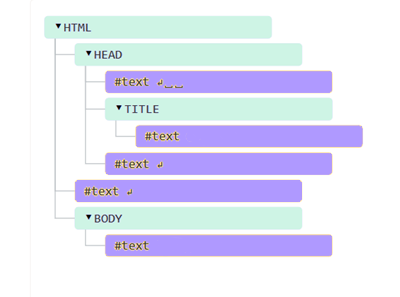


Рис.1.2. Дерево DOM.

### Типы узлов

**Теги** образуют узлы-элементы (element node). Естественным образом одни узлы вложены в другие. Структура дерева образована исключительно за счет них.

**Текст** внутри элементов образует текстовые узлы (text node), обозначенные как #text. Текстовый узел содержит исключительно строку текста и не может иметь потомков, то есть он всегда на самом нижнем уровне.

Обратите внимание на специальные символы в текстовых узлах:

перевод строки: ↵

пробел: ␣

Пробелы и переводы строки – это тоже текст, полноправные символы, которые учитываются в DOM.

В частности, в примере выше тег <html> содержит не только узлы-элементы <head> и <body>, но и #text (пробелы, переводы строки) между ними.

Впрочем, как раз на самом верхнем уровне из этого правила есть исключения: пробелы до <head> по стандарту игнорируются, а любое содержимое после </body> не создаёт узла, браузер переносит его внутрь, в конец body.

При чтении неверного HTML браузер автоматически корректирует его для показа и при построении DOM. В частности, всегда будет верхний тег <html>. Даже если в тексте нет – в DOM он будет, браузер создаст его самостоятельно. То же самое касается и тега <body>.

Например, если файл состоит из одного слова "Привет", то браузер автоматически обернёт его в <html> и <body>.

При генерации DOM браузер самостоятельно обрабатывает ошибки в документе, закрывает теги и так далее (пример 1.19 и рисунок 1.3).

**Пример 1.19.** Код до обработки ошибок

<p>Привет

<li>Мама

<li>и

<li>Папа

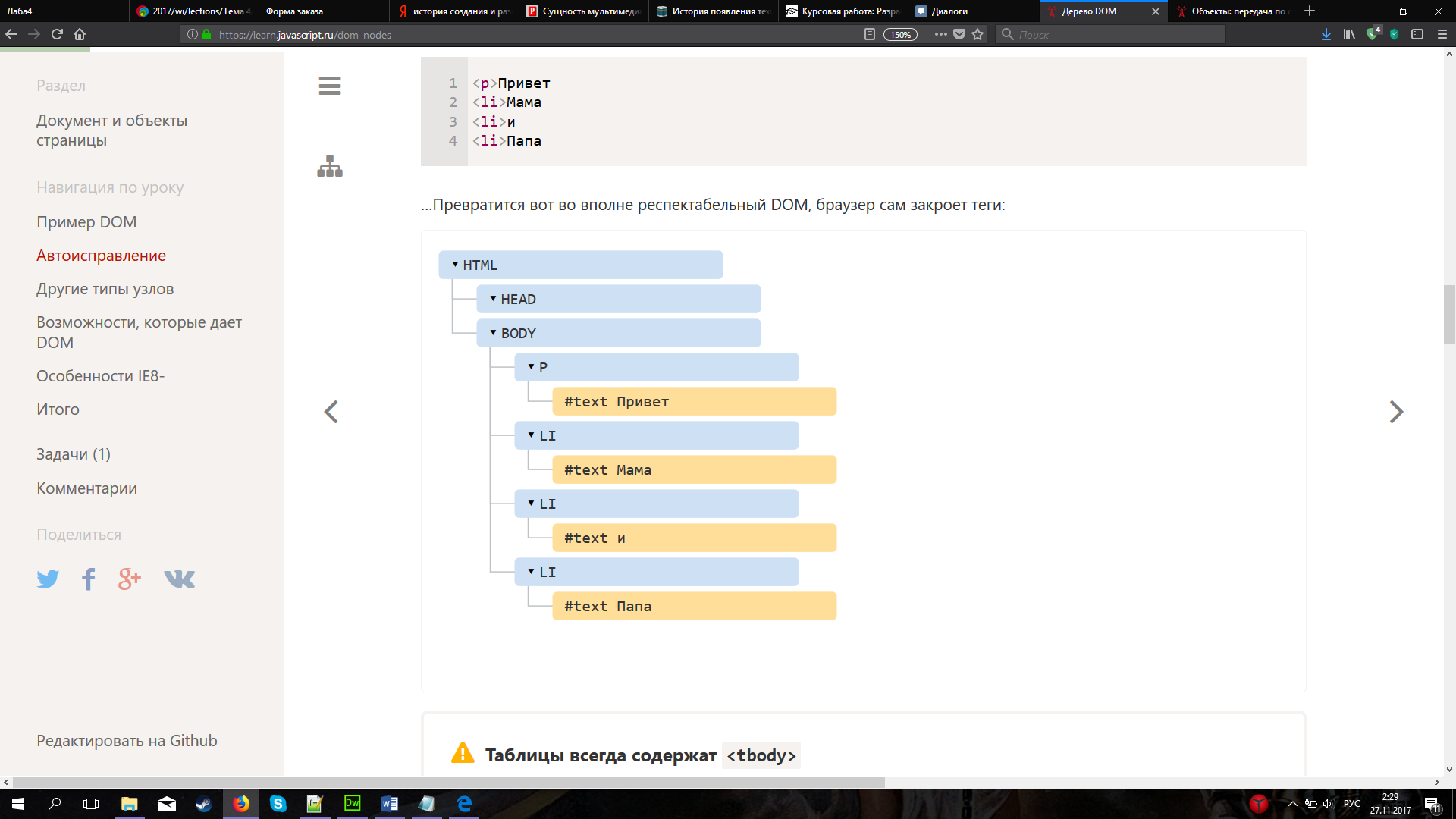


Рис.1.3. Пример 1.2 после автоисправления.

### Навигация по DOM-элементам

DOM позволяет делать что угодно с HTML-элементом и его содержимым, но для этого нужно сначала нужный элемент получить.

Доступ к DOM начинается с объекта document. Из него можно добраться до любых узлов.

**Точки входа**

Первая точка входа – document.documentElement. Это свойство ссылается на DOM-объект для тега <html>.

<BODY> = document.body

Вторая точка входа – document.body, который соответствует тегу <body>.

[**Дети: childNodes, firstChild, lastChild**](https://learn.javascript.ru/traversing-dom#дети-childnodes-firstchild-lastchild)

Здесь и далее мы будем использовать два принципиально разных термина.

**Дочерние элементы** (или дети) – элементы, которые лежат непосредственно внутри данного. Например, внутри <HTML> обычно лежат <HEAD> и <BODY>.

Потомки – все элементы, которые лежат внутри данного, вместе с их детьми, детьми их детей и так далее. То есть, всё поддерево DOM.

Псевдо-массив **childNodes** хранит все дочерние элементы, включая текстовые. Пример 1.20 покажет последовательно вывенные дочерние элементы document.b:

**Пример 1.20.** Дочерние элементы

<!DOCTYPE HTML>

<html>

<body>

<div>Начало</div>

<ul>

<li>Информация</li>

</ul>

<div>Конец</div>

<script>

for (var i = 0; i < document.body.childNodes.length; i++)

{

alert( document.body.childNodes[i] ); // Text, DIV, Text, UL, ..., SCRIPT

}

</script>

...

</body>

</html>

Обратим внимание на маленькую деталь. Если запустить пример выше, то последним будет выведен элемент <script>. На самом-то деле в документе есть ещё текст (обозначенный троеточием), но на момент выполнения скрипта браузер ещё до него не дошёл. Пробельный узел будет в итоговом документе, но его еще нет на момент выполнения скрипта.

Свойства **firstChild** и **lastChild** обеспечивают быстрый доступ к первому и последнему элементу. При наличии дочерних узлов всегда верно:

elem.childNodes[0] === elem.firstChild

elem.childNodes[elem.childNodes.length - 1] === elem.lastChild

DOM-коллекции, такие как childNodes и другие, которые мы увидим далее, не являются JavaScript-массивами.

В них нет методов массивов, таких как forEach, map, push, pop и других.

var elems = document.documentElement.childNodes;

elems.forEach(function(elem) { // нет такого метода!

/\* ... \*/

});

Именно поэтому childNodes и называют «коллекция» или «псевдомассив».

Можно для перебора коллекции использовать обычный цикл for(var i=0; i<elems.length; i++) ... Но что делать, если уж очень хочется воспользоваться методами массива?

Это возможно, основных варианта два:

1. Применить метод массива через call/apply:

var elems = document.documentElement.childNodes;

[].forEach.call(elems, function(elem) {

alert( elem ); // HEAD, текст, BODY

});

1. При помощи Array.prototype.slice сделать из коллекции массив. Обычно вызов arr.slice(a,b) делает новый массив и копирует туда элементы arr с индексами от a до b-1 включительно. Если же вызвать его без аргументов arr.slice(), то он делает новый массив и копирует туда все элементы arr.

Это работает и для коллекции:

var elems = document.documentElement.childNodes;

elems = Array.prototype.slice.call(elems); // теперь elems - массив

elems.forEach(function(elem) {

alert( elem.tagName ); // HEAD, текст, BODY

});

Нельзя перебирать коллекцию через for..in, ранее мы говорили, что не рекомендуется использовать для перебора массива цикл for..in.

Коллекции – наглядный пример, почему нельзя. Они похожи на массивы, но у них есть свои свойства и методы, которых в массивах нет.

В реальном коде нам нужны только элементы, мы же будем работать с ними, а служебные свойства – не нужны. Поэтому желательно использовать

for(var i=0; i<elems.length; i++).

**Соседи и родители**

Доступ к элементам слева и справа данного можно получить по ссылкам previousSibling / nextSibling. Родитель доступен через parentNode. Если долго идти от одного элемента к другому, то рано или поздно дойдёшь до корня DOM, то есть до document.documentElement, а затем и document.

**Навигация по элементам**

Навигационные ссылки, описанные выше, равно касаются всех узлов в документе. В частности, в childNodes сосуществуют и текстовые узлы и узлы-элементы и узлы-комментарии, если есть, но для большинства задач текстовые узлы нам не интересны. Поэтому посмотрим на дополнительный набор ссылок (рис.1.4), которые их не учитывают:

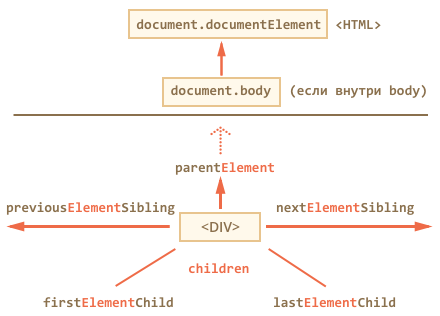


Рис.1.4. Ссылки, не учитывающие текстовые узлы.

Эти ссылки похожи на те, что раньше, только в ряде мест стоит слово Element:

* children – только дочерние узлы-элементы, то есть соответствующие тегам.
* firstElementChild, lastElementChild – соответственно, первый и последний дети-элементы.
* previousElementSibling, nextElementSibling – соседи-элементы.
* parentElement – родитель-элемент.

**Обход таблиц**

У конкретных элементов DOM могут быть свои дополнительные ссылки для большего удобства навигации. Рассмотрим таблицу, так как это важный частный случай и просто для примера.

TABLE

table.rows – коллекция строк TR таблицы.

parentElement – родитель-элемент.

table.caption/tHead/tFoot– ссылки на элементы таблицы CAPTION,HEAD,TFOOT

table.tBodies– коллекция элементов таблицы TBODY, по спецификации их может быть несколько.

THEAD/TFOOT/TBODY

tbody.rows – коллекция строк TR секции.

TR

tr.cells – коллекция ячеек TD/TH

tr.sectionRowIndex – номер строки в текущей секции THEAD/BODY

tr.rowIndex– номер строки в таблице

TD/TH

td.cellIndex – номер ячейки в строке

### Поиск

Прямая навигация от родителя к потомку удобна, если элементы рядом. А если нет? Как достать произвольный элемент откуда-то из глубины документа? Для этого в DOM есть дополнительные методы поиска – по id или document.getElementById.

Если элементу назначен специальный атрибут **id**, то можно получить его прямо по переменной с именем из значения id. Рассмотрим пример 1.21.

**Пример 1.21.** Получение атрибута по его id.

<div id="content-holder">

<div id="content">Элемент</div>

</div>

<script>

alert( content ); // DOM-элемент

alert( window['content-holder'] ); // в имени дефис, поэтому через [...]

</script>

Более правильной и общепринятой практикой является доступ к элементу вызовом **document.getElementById** ("идентификатор").

<div id="content">Выделим этот элемент</div>

<script>

var elem = document.getElementById('content');

elem.style.background = 'red';

alert( elem == content ); // true

content.style.background = ""; // один и тот же элемент

</script>

**По имени тега**

Метод **elem.getElementsByTagName(tag)** ищет все элементы с заданным тегом tag внутри элемента elem и возвращает их в виде списка, регистр тега не имеет значения (пример 1.22).

**Пример 1.22.** Пример применения метода elem.getElementsByTagName.

// получить все div-элементы

var elements = document.getElementsByTagName('div');

**По имени атрибута**

Вызов document.getElementsByName(name) позволяет получить все элементы с данным атрибутом name. Данный пример позволяет получить все элементы с именем age:

var elems = document.getElementsByName('age');

**По классу**

Вызов elem.getElementsByClassName(className) возвращает коллекцию элементов с классом className. Находит элемент и в том случае, если у него несколько классов, а искомый – один из них. Пример 1.23 возвращает коллекцию элементов с классом article.

**Пример 1.23.** Возвращение коллекции элементов с классом article

<div class="article">Статья</div>

<div class="long article">Длинная статья</div>

<script>

var articles = document.getElementsByClassName('article');

alert( articles.length ); // 2, найдёт оба элемента

</script>

Как и getElementsByTagName, этот метод может быть вызван и в контексте DOM-элемента, и в контексте документа.

**По селектору**

Вызов elem.querySelectorAll(CSS) возвращает все элементы внутри elem, удовлетворяющие CSS-селектору CSS.

Это один из самых часто используемых и полезных методов при работе с DOM. Рассмотрим пример 1.24:

**Пример 1.24.** Вызов метода elem.querySelectorAll.

<ul>

<li>Этот</li>

<li>тест</li>

</ul>

<ul>

<li>полностью</li>

<li>пройден</li>

</ul>

<script>

var elements = document.querySelectorAll('ul > li:last-child');

for (var i = 0; i < elements.length; i++) {

alert( elements[i].innerHTML ); // "тест", "пройден"

}

</script>

**Проверка условия**

Метод elem.matches(CSS) ничего не ищет, а проверяет, удовлетворяет ли elem селектору CSS. Он возвращает true либо false. Не поддерживается в IE8-.

Этот метод бывает полезным, когда мы перебираем элементы (в массиве или по обычным навигационным ссылкам) и пытаемся отфильтровать те из них, которые нам интересны (пример 1.25).

**Пример 1.25.** Применение метода elem.matches.

<a href="http://example.com/file.zip">...</a>

<a href="http://ya.ru">...</a>

<script>

var elems = document.body.children;

for (var i = 0; i < elems.length; i++) {

if (elems[i].matches('a[href$="zip"]')) {

alert( "Ссылка на архив: " + elems[i].href );

}

}

</script>

**Поиск вверх**

Метод elem.closest(CSS) ищет ближайший элемент выше по иерархии DOM, подходящий под CSS-селектор CSS. Сам элемент тоже включается в поиск. Иначе говоря, метод closest бежит от текущего элемента вверх по цепочке родителей и проверяет, подходит ли элемент под указанный CSS-селектор. Если подходит – останавливается и возвращает его. Он самый новый из методов, поэтому старые браузеры его слабо поддерживают.

**Пример 1.26.** Применение метода elem.closest.

<ul>

<li class="chapter">Глава I

<ul>

<li class="subchapter">Глава <span class="num">1.1</span></li>

<li class="subchapter">Глава <span class="num">1.2</span></li>

</ul>

</li>

</ul>

<script>

var numberSpan = document.querySelector('.num');

// ближайший элемент сверху подходящий под селектор li

alert(numberSpan.closest('li').className) // subchapter

// ближайший элемент сверху подходящий под селектор .chapter

alert(numberSpan.closest('.chapter').tagName) // LI

// ближайший элемент сверху, подходящий под селектор span

// это сам numberSpan, так как поиск включает в себя сам элемент

alert(numberSpan.closest('span') === numberSpan) // true

</script>

Сравнительная таблица поисковых методов:

**Таблица 1.6.** Сравнительная таблица поиска.

|  |  |  |  |
| --- | --- | --- | --- |
| Метод | Ищет по | Ишет внутри элемента? | Поддержка |
| getElementById | Id | Нет | Везде |
| getElementsByName | Name | Нет | Везде |
| getElementsByTagName | Тег или ‘\*’ | Да | Везде |
| getElementsByClassName | Классу | Да | Кроме IE8 |
| querySelectorAll | CSS-селектор | Да | везде |

### Свойства

Классы DOM образуют иерархию. Основной объект в ней: Node, от которого наследуют остальные (рис. 1.5):

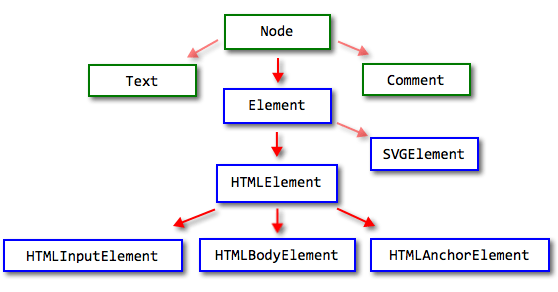


Рис.1.5. Наглядное изображение дерева Node.

**Тип узла nodeType**

Тип узла содержится в его свойстве nodeType. Как правило, мы работаем всего с двумя типами узлов:

1. элемент;
2. текстовый узел.

На самом деле типов узлов гораздо больше. Строго говоря, их 12, и они описаны в спецификации.

В частности, тип «Элемент» ELEMENT\_NODE имеет номер 1, а «Текст» TEXT\_NODE – номер 3.

Тип узла можно только читать, изменить его невозможно.

**nodeName и tagName**

Существует целых два свойства: nodeName и tagName, которые содержат название (тег) элемента узла. Название HTML-тега всегда находится в верхнем регистре. Рассмотрим пример 1.27 для document.body:

**Пример 1.27.** Применение свойств nodeName и tagName.

alert( document.body.nodeName ); // BODY

alert( document.body.tagName ); // BODY

Какая разница между nodeName и tagName? Свойство tagName есть только у элементов Element (в IE8-также у комментариев, но это ошибка в браузере).

Свойство nodeName определено для любых узлов Node, для элементов оно равно tagName, а для не-элементов обычно содержит строку с типом узла

Таким образом, при помощи tagName мы можем работать только с элементами, а nodeName может что-то сказать и о других типах узлов.

Свойство innerHTML позволяет получить HTML-содержимое элемента в виде строки. В innerHTML можно и читать, и писать. Пример 1.28 выведет на экран все содержимое document.body, а затем заменит его на другое:

**Пример 1.28.** Применение свойства innerHTML.

<body>

<p>Параграф</p>

<div>Div</div>

<script>

alert( document.body.innerHTML ); // читаем текущее содержимое

document.body.innerHTML = 'Новый BODY!'; // заменяем содержимое

</script>

</body>

Свойство outerHTML содержит HTML элемента целиком (пример 1.29).

**Пример 1.29.** Применение свойства outerHTML.

<div>Привет <b>Мир</b></div>

<script>

var div = document.body.children[0];

alert( div.outerHTML ); // <div>Привет <b>Мир</b></div>

</script>

Изменить outerHTML элемента невозможно. Свойство innerHTML есть только у узлов-элементов. Содержимое других узлов, например, текстовых или комментариев, доступно на чтение и запись через свойство data. Его тоже можно читать и обновлять.

**Пример 1.30.** Применение свойства data.

<body>

Привет

<!-- Комментарий -->

<script>

for (var i = 0; i < document.body.childNodes.length; i++) {

alert( document.body.childNodes[i].data );

}

</script>

Пока

</body>

Если вы запустите этот пример, то увидите, как выводятся последовательно:

«Привет» – это содержимое первого узла(текстового);

комментарий – это содержимое второго узла(комментария);

пробелы – это содержимое небольшого пробельного узла после комментария до скрипта;

Undefined – далее цикл дошел до <script>, но это узел – элемент, у него нет data.

textContent дает возможность записать текст в элемент, причём именно как текст. В примере 1.31 имя посетителя попадёт в первый div как innerHTML, а во второй – как текст:

**Пример 1.31.** Применение свойства textContent.

<div></div>

<div></div>

<script>

var name = prompt("Введите имя?", "<b>Винни-пух</b>");

document.body.children[0].innerHTML = name;

document.body.children[1].textContent = name;

</script>

При запуске примера мы увидим, что в первый DIV текст от посетителя вставился именно как HTML, то есть теги стали именно тегами, а во второй – как обычный текст.

Вряд ли мы действительно хотим, чтобы посетители вставляли в наш сайт произвольный HTML-код. Присваивание через textContent – один из способов от этого защититься. У DOM-узлов есть и другие свойства, зависящие от типа, например:

value – значение для INPUT, SELECT или TEXTAREA;

id – идентификатор;

href – адрес ссылки.

**Пример 1.32.** Применение свойств value и id.

<input type="text" id="elem" value="значение">

<script>

var input = document.body.children[0];

alert( input.type ); // "text"

alert( input.id ); // "elem"

alert( input.value ); // значение

</script>

### Атрибуты и свойства DOM

Когда браузер загружает страницу, он «читает» HTML и генерирует из него DOM-объекты. Для узлов-элементов большинство стандартных HTML-атрибутов автоматически становятся свойствами DOM-объектов.

Например, для такого тега <body id="page"> у DOM-объекта будет такое свойство body.id="page".

Но преобразование атрибута в свойство происходит не один-в-один! Рассмотрим различия этих двух понятий, чтобы посмотреть, как работать с ними, когда они одинаковые и когда разные.

**[DOM-свойства](file:///C:\\Users\\user\\Desktop\\Web-интерфейсы\\Часть%202%20материалы\\1.4%20Атрибуты%20и%20свойства.html" \l "dom-svoystva)**

Ранее мы уже видели встроенные DOM-свойства. Их много. Но технически нас никто не ограничивает, и если этого мало – мы можем добавить своё собственное свойство.

DOM-узлы – это обычные объекты JavaScript. Мы можем их изменять. Например, создадим новое свойство для document.body:

document.body.myData = {

name: 'Caesar',

title: 'Imperator'

};

alert(document.body.myData.title); // Imperator

Мы можем добавить и метод:

document.body.sayTagName = function() {

alert(this.tagName);

};

document.body.sayTagName(); // BODY (значением "this" в этом методе будет document.body)

Также можно изменять встроенные прототипы, такие как Element.prototype и добавлять новые методы ко всем элементам:

Element.prototype.sayHi = function() {

alert(`Hello, I'm ${this.tagName}`);

};

document.documentElement.sayHi(); // Hello, I'm HTML

document.body.sayHi(); // Hello, I'm BODY

Итак, DOM-свойства и методы ведут себя так же, как и обычные объекты JavaScript:

1. Им можно присвоить любое значение.
2. Они регистрозависимы (нужно писать elem.nodeType, не elem.NoDeTyPe).

**[HTML-атрибуты](file:///C:\\Users\\user\\Desktop\\Web-интерфейсы\\Часть%202%20материалы\\1.4%20Атрибуты%20и%20свойства.html" \l "html-atributy)**

В HTML у тегов могут быть атрибуты. Когда браузер парсит HTML, чтобы создать DOM-объекты для тегов, он распознаёт стандартные атрибуты и создаёт DOM-свойства для них. Таким образом, когда у элемента есть id или другой стандартный атрибут, создаётся соответствующее свойство. Но этого не происходит, если атрибут нестандартный.

Например:

<body id="test" something="non-standard">

<script>

alert(document.body.id); // test

// нестандартный атрибут не преобразуется в свойство

alert(document.body.something); // undefined

</script>

</body>

Пожалуйста, учтите, что стандартный атрибут для одного тега может быть нестандартным для другого. Например, атрибут "type" является стандартным для элемента <input> (HTMLInputElement), но не является стандартным для <body> (HTMLBodyElement). Стандартные атрибуты описаны в спецификации для соответствующего класса элемента.

Мы можем увидеть это на примере ниже:

<body id="body" type="...">

<input id="input" type="text">

<script>

alert(input.type); // text

alert(body.type); // undefined: DOM-свойство не создалось, потому что оно нестандартное

</script>

</body>

Таким образом, для нестандартных атрибутов не будет соответствующих DOM-свойств. Есть ли способ получить такие атрибуты?

Конечно. Все атрибуты доступны с помощью следующих методов:

elem.hasAttribute(name) – проверяет наличие атрибута.

elem.getAttribute(name) – получает значение атрибута.

elem.setAttribute(name, value) – устанавливает значение атрибута.

elem.removeAttribute(name) – удаляет атрибут.

Этим методы работают именно с тем, что написано в HTML.

Кроме этого, получить все атрибуты элемента можно с помощью свойства elem.attributes: коллекция объектов, которая принадлежит ко встроенному классу [Attr](https://dom.spec.whatwg.org/#attr) со свойствами name и value.

Вот демонстрация чтения нестандартного свойства:

<body something="non-standard">

<script>

alert(document.body.getAttribute('something')); // non-standard

</script>

</body>

У HTML-атрибутов есть следующие особенности:

* их имена регистронезависимы (id то же самое, что и ID);
* их значения всегда являются строками.

Расширенная демонстрация работы с атрибутами:

<body>

<div id="elem" about="Elephant"></div>

<script>

alert( elem.getAttribute('About') ); // (1) 'Elephant', чтение

elem.setAttribute('Test', 123); // (2), запись

alert( elem.outerHTML ); // (3), посмотрим, есть ли атрибут в HTML (да)

for (let attr of elem.attributes) { // (4) весь список

alert( `${attr.name} = ${attr.value}` );

}

</script>

</body>

Обратите внимание на следующее:

1. getAttribute('About') – здесь первая буква заглавная, а в HTML – строчная. Но это не важно: имена атрибутов регистронезависимы.
2. Мы можем присвоить что угодно атрибуту, но это станет строкой. Поэтому в этой строчке мы получаем значение "123".
3. Все атрибуты, в том числе те, которые мы установили, видны в outerHTML.
4. Коллекция attributes является перебираемой. В ней есть все атрибуты элемента (стандартные и нестандартные) в виде объектов со свойствами name и value.

**Синхронизация между атрибутами и свойствами**

Когда стандартный атрибут изменяется, соответствующее свойство автоматически обновляется. Это работает и в обратную сторону (за некоторыми исключениями).

В примере ниже id модифицируется как атрибут, и можно увидеть, что свойство также изменено. То же самое работает и в обратную сторону:

<input>

<script>

let input = document.querySelector('input');

// атрибут => свойство

input.setAttribute('id', 'id');

alert(input.id); // id (обновлено)

// свойство => атрибут

input.id = 'newId';

alert(input.getAttribute('id')); // newId (обновлено)

</script>

Но есть и исключения, например, input.value синхронизируется только в одну сторону – атрибут → значение, но не в обратную:

<input>

<script>

let input = document.querySelector('input');

// атрибут => значение

input.setAttribute('value', 'text');

alert(input.value); // text

// свойство => атрибут

input.value = 'newValue';

alert(input.getAttribute('value')); // text (не обновилось!)

</script>

В примере выше:

1. Изменение атрибута value обновило свойство.
2. Но изменение свойства не повлияло на атрибут.

Иногда эта «особенность» может пригодиться, потому что действия пользователя могут приводить к изменениям value, и если после этого мы захотим восстановить «оригинальное» значение из HTML, оно будет в атрибуте.

**[DOM-свойства типизированы](file:///C:\\Users\\user\\Desktop\\Web-интерфейсы\\Часть%202%20материалы\\1.4%20Атрибуты%20и%20свойства.html" \l "dom-svoystva-tipizirovany)**

DOM-свойства не всегда являются строками. Например, свойство input.checked (для чекбоксов) имеет логический тип:

<input id="input" type="checkbox" checked> checkbox

<script>

alert(input.getAttribute('checked')); // значение атрибута: пустая строка

alert(input.checked); // значение свойства: true

</script>

Есть и другие примеры. Атрибут style – строка, но свойство style является объектом:

<div id="div" style="color:red;font-size:120%">Hello</div>

<script>

// строка

alert(div.getAttribute('style')); // color:red;font-size:120%

// объект

alert(div.style); // [object CSSStyleDeclaration]

alert(div.style.color); // red

</script>

Хотя большинство свойств, всё же, строки. При этом некоторые из них, хоть и строки, могут отличаться от атрибутов. Например, DOM-свойство href всегда содержит полный URL, даже если атрибут содержит относительный URL или просто #hash.

Ниже пример:

<a id="a" href="#hello">link</a>

<script>

// атрибут

alert(a.getAttribute('href')); // #hello

// свойство

alert(a.href ); // полный URL в виде http://site.com/page#hello

</script>

Если же нужно значение href или любого другого атрибута в точности, как оно записано в HTML, можно воспользоваться getAttribute.

**[Нестандартные атрибуты, dataset](file:///C:\\Users\\user\\Desktop\\Web-интерфейсы\\Часть%202%20материалы\\1.4%20Атрибуты%20и%20свойства.html" \l "nestandartnye-atributy-dataset)**

При написании HTML мы используем много стандартных атрибутов. Но что насчёт нестандартных, пользовательских? Для чего они нужны?

Иногда нестандартные атрибуты используются для передачи пользовательских данных из HTML в JavaScript, или чтобы «помечать» HTML-элементы для JavaScript. Как тут:

<!-- пометить div, чтобы показать здесь поле "name" -->

<div show-info="name"></div>

<!-- а здесь возраст "age" -->

<div show-info="age"></div>

<script>

// код находит элемент с пометкой и показывает запрошенную информацию

let user = {

name: "Pete",

age: 25

};

for(let div of document.querySelectorAll('[show-info]')) {

// вставить соответствующую информацию в поле

let field = div.getAttribute('show-info');

div.innerHTML = user[field]; // сначала Pete в name, потом 25 в age

}

</script>

Также они могут быть использованы, чтобы стилизовать элементы. Например, здесь для состояния заказа используется атрибут order-state:

<style>

/\* стили зависят от пользовательского атрибута "order-state" \*/

.order[order-state="new"] {

color: green;

}

.order[order-state="pending"] {

color: blue;

}

.order[order-state="canceled"] {

color: red;

}

</style>

<div class="order" order-state="new">

A new order.

</div>

<div class="order" order-state="pending">

A pending order.

</div>

<div class="order" order-state="canceled">

A canceled order.

</div>

Почему атрибут может быть предпочтительнее таких классов, как .order-state-new, .order-state-pending, order-state-canceled? Это потому, что атрибутом удобнее управлять. Состояние может быть изменено достаточно просто:

// немного проще, чем удаление старого/добавление нового класса

div.setAttribute('order-state', 'canceled');

Но с пользовательскими атрибутами могут возникнуть проблемы. Что если мы используем нестандартный атрибут для наших целей, а позже он появится в стандарте и будет выполнять какую-то функцию? В HTML появляется больше атрибутов, чтобы удовлетворить потребности разработчиков. В этом случае могут возникнуть неожиданные эффекты.

Чтобы избежать конфликтов, существуют атрибуты вида [data-\*](https://html.spec.whatwg.org/#embedding-custom-non-visible-data-with-the-data-*-attributes).

Все атрибуты, начинающиеся с префикса «data-», зарезервированы для использования программистами. Они доступны в свойстве dataset.

Например, если у elem есть атрибут "data-about", то обратиться к нему можно как elem.dataset.about:

<body data-about="Elephants">

<script>

alert(document.body.dataset.about); // Elephants

</script>

Атрибуты, состоящие из нескольких слов, к примеру data-order-state, становятся свойствами, записанными с помощью верблюжьей нотации: dataset.orderState.

Вот переписанный пример «состояния заказа»:

<style>

.order[data-order-state="new"] {

color: green;

}

.order[data-order-state="pending"] {

color: blue;

}

.order[data-order-state="canceled"] {

color: red;

}

</style>

<div id="order" class="order" data-order-state="new">

A new order.

</div>

<script>

// чтение

alert(order.dataset.orderState); // new

// изменение

order.dataset.orderState = "pending"; // (\*)

</script>

Использование data-\* атрибутов – валидный, безопасный способ передачи пользовательских данных. Причем можно не только читать, но и изменять data-атрибуты. Тогда CSS обновит представление соответствующим образом: в примере выше последняя строка (\*) меняет цвет на синий.

### Изменения в DOM

Модификации DOM – это ключ к созданию «живых» страниц. Рассмотрим методы на примере – а именно, добавим на страницу сообщение, которое будет выглядеть получше, чем alert.

Вот такое:

<style>

.alert {

padding: 15px;

border: 1px solid #d6e9c6;

border-radius: 4px;

color: #3c763d;

background-color: #dff0d8;

}

</style>

<div class="alert">

<strong>Всем привет!</strong> Вы прочитали важное сообщение.

</div>

Это был пример HTML. Теперь давайте создадим такой же div, используя JavaScript (предполагаем, что стили в HTML или во внешнем CSS-файле).

DOM-узел можно создать двумя методами:

document.createElement(tag)

Создаёт новый элемент с заданным тегом:

let div = document.createElement('div');

document.createTextNode(text)

Создаёт новый текстовый узел с заданным текстом:

let textNode = document.createTextNode('А вот и я');

**[Создание сообщения](file:///C:\\Users\\user\\Desktop\\Web-%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D1%8B\\%D0%A7%D0%B0%D1%81%D1%82%D1%8C%202%20%D0%BC%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D1%8B\\1.4.%20%D0%98%D0%B7%D0%BC%D0%B5%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5%20DOM.html" \l "sozdanie-soobscheniya)**

В нашем случае сообщение – это div с классом alert и HTML в нём:

let div = document.createElement('div');

div.className = "alert";

div.innerHTML = "<strong>Всем привет!</strong> Вы прочитали важное сообщение.";

Мы создали элемент, но пока он только в переменной. Мы не можем видеть его на странице, поскольку он не является частью документа. Чтобы наш div появился, нам нужно вставить его где-нибудь в document, например, в document.body. Для этого есть метод append, в нашем случае: document.body.append(div).

Вот полный пример:

<style>

.alert {

padding: 15px;

border: 1px solid #d6e9c6;

border-radius: 4px;

color: #3c763d;

background-color: #dff0d8;

}

</style>

<script>

let div = document.createElement('div');

div.className = "alert";

div.innerHTML = "<strong>Всем привет!</strong> Вы прочитали важное сообщение.";

document.body.append(div);

</script>

Вот методы для различных вариантов вставки:

* node.append(...nodes or strings) – добавляет узлы или строки в конец node;
* node.prepend(...nodes or strings) – вставляет узлы или строки в начало node;
* node.before(...nodes or strings) – вставляет узлы или строки до node;
* node.after(...nodes or strings) – вставляет узлы или строки после node;
* node.replaceWith(...nodes or strings) – заменяет node заданными узлами или строками.

Вот пример использования этих методов, чтобы добавить новые элементы в список и текст до/после него:

<ol id="ol">

<li>0</li>

<li>1</li>

<li>2</li>

</ol>

<script>

ol.before('before'); // вставить строку "before" перед <ol>

ol.after('after'); // вставить строку "after" после <ol>

let liFirst = document.createElement('li');

liFirst.innerHTML = 'prepend';

ol.prepend(liFirst); // вставить liFirst в начало <ol>

let liLast = document.createElement('li');

liLast.innerHTML = 'append';

ol.append(liLast); // вставить liLast в конец <ol>

</script>

Наглядная иллюстрация того, куда эти методы вставляют:

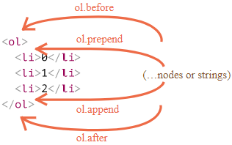


Рис. 1.6. Методы различных вариантов вставки сообщения.

Итоговый список будет таким:

before

<ol id="ol">

<li>prepend</li>

<li>0</li>

<li>1</li>

<li>2</li>

<li>append</li>

</ol>

after

Эти методы могут вставлять несколько узлов и текстовых фрагментов за один вызов. Например, здесь вставляется строка и элемент:

<div id="div"></div>

<script>

div.before('<p>Привет</p>', document.createElement('hr'));

</script>

Весь текст вставляется как текст, поэтому финальный HTML будет:

&lt;p&gt;Привет&lt;/p&gt;

<hr>

<div id="div"></div>

Другими словами, строки вставляются безопасным способом, как делает это elem.textContent. Поэтому эти методы могут использоваться только для вставки DOM-узлов или текстовых фрагментов. А что, если мы хотим вставить HTML именно «как html», со всеми тегами и прочим, как делает это elem.innerHTML?

**[insertAdjacentHTML/Text/Element](file:///C:\\Users\\user\\Desktop\\Web-%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D1%8B\\%D0%A7%D0%B0%D1%81%D1%82%D1%8C%202%20%D0%BC%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D1%8B\\1.4.%20%D0%98%D0%B7%D0%BC%D0%B5%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5%20DOM.html" \l "insertadjacenthtml-text-element)**

С этим может помочь другой, довольно универсальный метод: elem.insertAdjacentHTML(where, html).

Первый параметр – это специальное слово, указывающее, куда по отношению к elem производить вставку. Значение должно быть одним из следующих:

* "beforebegin" – вставить html непосредственно перед elem;
* "afterbegin" – вставить html в начало elem;
* "beforeend" – вставить html в конец elem;
* "afterend" – вставить html непосредственно после elem.

Второй параметр – это HTML-строка, которая будет вставлена именно «как HTML», например:

<div id="div"></div>

<script>

div.insertAdjacentHTML('beforebegin', '<p>Привет</p>');

div.insertAdjacentHTML('afterend', '<p>Пока</p>');

</script>

приведёт к:

<p>Привет</p>

<div id="div"></div>

<p>Пока</p>

Так мы можем добавлять произвольный HTML на страницу. Варианты вставки:



Рис. 1.7. Варианты вставки кода на страницу.

Мы можем легко заметить сходство между этой и предыдущей картинкой. Точки вставки фактически одинаковые, но этот метод вставляет HTML.

У метода есть два подобных метода:

* elem.insertAdjacentText(where, text) – такой же синтаксис, но строка text вставляется «как текст», вместо HTML;
* elem.insertAdjacentElement(where, elem) – такой же синтаксис, но вставляет элемент elem.

Они существуют, в основном, чтобы унифицировать синтаксис. На практике часто используется только insertAdjacentHTML. Потому что для элементов и текста у нас есть методы append/prepend/before/after – их быстрее написать, и они могут вставлять как узлы, так и текст. Альтернативный вариант показа сообщения:

<style>

.alert {

padding: 15px;

border: 1px solid #d6e9c6;

border-radius: 4px;

color: #3c763d;

background-color: #dff0d8;

}

</style>

<script>

document.body.insertAdjacentHTML("afterbegin", `<div class="alert">

<strong>Всем привет!</strong> Вы прочитали важное сообщение.

</div>`);

</script>

**[Удаление узлов](file:///C:\\Users\\user\\Desktop\\Web-%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D1%8B\\%D0%A7%D0%B0%D1%81%D1%82%D1%8C%202%20%D0%BC%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D1%8B\\1.4.%20%D0%98%D0%B7%D0%BC%D0%B5%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5%20DOM.html" \l "udalenie-uzlov)**

Для удаления узла есть методы node.remove(). Например, сделаем так, чтобы наше сообщение удалялось через секунду:

<style>

.alert {

padding: 15px;

border: 1px solid #d6e9c6;

border-radius: 4px;

color: #3c763d;

background-color: #dff0d8;

}

</style>

<script>

let div = document.createElement('div');

div.className = "alert";

div.innerHTML = "<strong>Всем привет!</strong> Вы прочитали важное сообщение.";

document.body.append(div);

setTimeout(() => div.remove(), 1000);

</script>

Если нам нужно переместить элемент в другое место – нет необходимости удалять его со старого. Все методы вставки автоматически удаляют узлы со старых мест. Например, давайте поменяем местами элементы:

<div id="first">Первый</div>

<div id="second">Второй</div>

<script>

// нет необходимости вызывать метод remove

second.after(first); // берёт #second и после него вставляет #first

</script>

**[Клонирование узлов: cloneNode](file:///C:\\Users\\user\\Desktop\\Web-%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D1%8B\\%D0%A7%D0%B0%D1%81%D1%82%D1%8C%202%20%D0%BC%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D1%8B\\1.4.%20%D0%98%D0%B7%D0%BC%D0%B5%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5%20DOM.html" \l "klonirovanie-uzlov-clonenode)**

Как вставить ещё одно подобное сообщение? Мы могли бы создать функцию и поместить код туда. Альтернатива – клонировать существующий div и изменить текст внутри него (при необходимости). Иногда, когда у нас есть большой элемент, это может быть быстрее и проще.

Вызов elem.cloneNode(true) создаёт «глубокий» клон элемента – со всеми атрибутами и дочерними элементами. Если мы вызовем elem.cloneNode(false), тогда клон будет без дочерних элементов.

Пример копирования сообщения:

<style>

.alert {

padding: 15px;

border: 1px solid #d6e9c6;

border-radius: 4px;

color: #3c763d;

background-color: #dff0d8;

}

</style>

<div class="alert" id="div">

<strong>Всем привет!</strong> Вы прочитали важное сообщение.

</div>

<script>

let div2 = div.cloneNode(true); // клонировать сообщение

div2.querySelector('strong').innerHTML = 'Всем пока!'; // изменить клонированный элемент

div.after(div2); // показать клонированный элемент после существующего div

</script>

**[DocumentFragment](file:///C:\\Users\\user\\Desktop\\Web-%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D1%8B\\%D0%A7%D0%B0%D1%81%D1%82%D1%8C%202%20%D0%BC%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D1%8B\\1.4.%20%D0%98%D0%B7%D0%BC%D0%B5%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5%20DOM.html" \l "document-fragment)**

DocumentFragment является специальным DOM-узлом, который служит обёрткой для передачи списков узлов. Мы можем добавить к нему другие узлы, но когда мы вставляем его куда-то, он «исчезает», вместо него вставляется его содержимое.

Например, getListContent ниже генерирует фрагмент с элементами <li>, которые позже вставляются в <ul>:

<ul id="ul"></ul>

<script>

function getListContent() {

let fragment = new DocumentFragment();

for(let i=1; i<=3; i++) {

let li = document.createElement('li');

li.append(i);

fragment.append(li);

}

return fragment;

}

ul.append(getListContent()); // (\*)

</script>

Обратите внимание, что на последней строке с (\*) мы добавляем DocumentFragment, но он «исчезает», поэтому структура будет:

<ul>

<li>1</li>

<li>2</li>

<li>3</li>

</ul>

DocumentFragment редко используется. Зачем добавлять элементы в специальный вид узла, если вместо этого мы можем вернуть массив узлов? Переписанный пример:

<ul id="ul"></ul>

<script>

function getListContent() {

let result = [];

for(let i=1; i<=3; i++) {

let li = document.createElement('li');

li.append(i);

result.push(li);

}

return result;

}

ul.append(...getListContent()); // append + оператор "..." = друзья!

</script>

Мы упоминаем DocumentFragment в основном потому, что он используется в некоторых других областях, например, для элемента [template](https://learn.javascript.ru/template-element).

### Стили и классы

Как правило, существует два способа задания стилей для элемента:

1. Создать класс в CSS и использовать его: <div class="...">
2. Писать стили непосредственно в атрибуте style: <div style="...">.

JavaScript может менять и классы, и свойство style. Классы – всегда предпочтительный вариант по сравнению со style. Мы должны манипулировать свойством style только в том случае, если классы «не могут справиться».

Например, использование style является приемлемым, если мы вычисляем координаты элемента динамически и хотим установить их из JavaScript:

let top = /\* сложные расчёты \*/;

let left = /\* сложные расчёты \*/;

elem.style.left = left; // например, '123px', значение вычисляется во время работы скрипта

elem.style.top = top; // например, '456px'

В других случаях, например, чтобы сделать текст красным, добавить значок фона – описываем это в CSS и добавляем класс (JavaScript может это сделать). Это более гибкое и лёгкое в поддержке решение.

**[className и classList](file:///C:\\Users\\user\\Desktop\\Web-%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D1%8B\\%D0%A7%D0%B0%D1%81%D1%82%D1%8C%202%20%D0%BC%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D1%8B\\1.4%20%D0%A1%D1%82%D0%B8%D0%BB%D0%B8%20%D0%B8%20%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D1%8B.html" \l "classname-i-classlist)**

Изменение класса является одним из наиболее часто используемых действий в скриптах. Когда-то давно в JavaScript существовало ограничение: зарезервированное слово типа "class" не могло быть свойством объекта. Это ограничение сейчас отсутствует, но в то время было невозможно иметь свойство elem.class. Поэтому для классов было введено схожее свойство "className": elem.className соответствует атрибуту "class".

Например:

<body class="main page">

<script>

alert(document.body.className); // main page

</script>

</body>

Если мы присваиваем что-то elem.className, то это заменяет всю строку с классами. Иногда это то, что нам нужно, но часто мы хотим добавить/удалить один класс. Для этого есть другое свойство: elem.classList.

elem.classList – это специальный объект с методами для добавления/удаления одного класса, например:

<body class="main page">

<script>

// добавление класса

document.body.classList.add('article');

alert(document.body.className); // main page article

</script>

</body>

Так что мы можем работать как со строкой полного класса, используя className, так и с отдельными классами, используя classList. Выбираем тот вариант, который нам удобнее. Методы classList:

elem.classList.add/remove("class") – добавить/удалить класс.

elem.classList.toggle("class") – добавить класс, если его нет, иначе удалить.

elem.classList.contains("class") – проверка наличия класса, возвращает true/false.

Кроме того, classList является перебираемым, поэтому можно перечислить все классы при помощи for..of:

<body class="main page">

<script>

for (let name of document.body.classList) {

alert(name); // main, затем page

}

</script>

</body>

**[Element style](file:///C:\\Users\\user\\Desktop\\Web-%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D1%8B\\%D0%A7%D0%B0%D1%81%D1%82%D1%8C%202%20%D0%BC%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D1%8B\\1.4%20%D0%A1%D1%82%D0%B8%D0%BB%D0%B8%20%D0%B8%20%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D1%8B.html" \l "element-style)**

Свойство elem.style – это объект, который соответствует тому, что написано в атрибуте "style". Установка стиля elem.style.width="100px" работает так же, как наличие в атрибуте style строки width:100px.

Для свойства из нескольких слов используется camelCase:

background-color => elem.style.backgroundColor

z-index => elem.style.zIndex

border-left-width => elem.style.borderLeftWidth

Например:

document.body.style.backgroundColor = prompt('background color?', 'green');

Свойства с префиксом

Стили с браузерным префиксом, например, -moz-border-radius, -webKit-border-radius преобразуются по тому же принципу: дефис означает прописную букву, например:

button.style.MozBorderRadius = '5px';

button.style.WebKitBorderRadius = '5px';

**[Сброс стилей](file:///C:\\Users\\user\\Desktop\\Web-%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D1%8B\\%D0%A7%D0%B0%D1%81%D1%82%D1%8C%202%20%D0%BC%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D1%8B\\1.4%20%D0%A1%D1%82%D0%B8%D0%BB%D0%B8%20%D0%B8%20%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D1%8B.html" \l "sbros-stiley)**

Иногда нам нужно добавить свойство стиля, а потом, позже, убрать его.

Например, чтобы скрыть элемент, мы можем задать elem.style.display = "none".

Затем мы можем удалить свойство style.display, чтобы вернуться к первоначальному состоянию. Вместо delete elem.style.display мы должны присвоить ему пустую строку: elem.style.display = "".

// если мы запустим этот код, <body> "мигнёт"

document.body.style.display = "none"; // скрыть

setTimeout(() => document.body.style.display = "", 1000); // возврат к нормальному состоянию

Если мы установим в style.display пустую строку, то браузер применит CSS-классы и встроенные стили, как если бы такого свойства style.display вообще не было.

Обычно мы используем style.\* для присвоения индивидуальных свойств стиля. Нельзя установить список стилей как, например, div.style="color: red; width: 100px", потому что div.style – это объект, и он доступен только для чтения.

Для задания нескольких стилей в одной строке используется специальное свойство style.CSSText:

<div id="div">Button</div>

<script>

// можем даже устанавливать специальные флаги для стилей, например, "important"

div.style.CSSText=`color: red !important;

background-color: yellow;

width: 100px;

text-align: center;

`;

alert(div.style.CSSText);

</script>

Это свойство редко используется, потому что такое присваивание удаляет все существующие стили: оно не добавляет, а заменяет их. Можно ненароком удалить что-то нужное. Но его можно использовать, к примеру, для новых элементов, когда мы точно знаем, что не удалим существующий стиль.

То же самое можно сделать установкой атрибута: div.setAttribute('style', 'color: red...').

Не забывайте добавлять к значениям единицы измерения, иначе свойство не сработает.

Обратите внимание: браузер «распаковывает» свойство style.margin в последних строках и выводит style.marginLeft и style.marginTop из него.

**[Вычисленные стили: getComputedStyle](file:///C:\\Users\\user\\Desktop\\Web-%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D1%8B\\%D0%A7%D0%B0%D1%81%D1%82%D1%8C%202%20%D0%BC%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D1%8B\\1.4%20%D0%A1%D1%82%D0%B8%D0%BB%D0%B8%20%D0%B8%20%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D1%8B.html" \l "vychislennye-stili-getcomputedstyle)**

Итак, изменить стиль очень просто. Но как его прочитать?

Например, мы хотим знать размер, отступы, цвет элемента. Как это сделать?

Свойство style оперирует только значением атрибута "style", без учёта CSS-каскада. Поэтому, используя elem.style, мы не можем прочитать ничего, что приходит из классов CSS. Например, здесь style не может видеть отступы:

<head>

<style> body { color: red; margin: 5px } </style>

</head>

<body>

Красный текст

<script>

alert(document.body.style.color); // пусто

alert(document.body.style.marginTop); // пусто

</script>

</body>

Что, если нужно, скажем, увеличить отступ на 20px? Для начала нужно его текущее значение получить, для этого есть метод: getComputedStyle. Синтаксис:

getComputedStyle(element, [pseudo])

element − элемент, значения для которого нужно получить

pseudo− указывается, если нужен стиль псевдоэлемента, например ::before. Пустая строка или отсутствие аргумента означают сам элемент. Результат вызова – объект со стилями, похожий на elem.style, но с учётом всех CSS-классов.

Например:

<head>

<style> body { color: red; margin: 5px } </style>

</head>

<body>

<script>

let computedStyle = getComputedStyle(document.body);

// сейчас мы можем прочитать отступ и цвет

alert( computedStyle.marginTop ); // 5px

alert( computedStyle.color ); // rgb(255, 0, 0)

</script>

</body>

**Вычисленное (computed) и окончательное (resolved) значения**

Есть две концепции в [CSS](https://drafts.csswg.org/cssom/#resolved-values):

1. Вычисленное (computed) значение – это то, которое получено после применения всех CSS-правил и CSS-наследования. Например, height:1em или font-size:125%.
2. Окончательное ([resolved](https://drafts.csswg.org/cssom/#resolved-values)) значение – непосредственно применяемое к элементу. Значения 1em или 125% являются относительными. Браузер берёт вычисленное значение и делает все единицы измерения фиксированными и абсолютными, например, height:20px или font-size:16px. Для геометрических свойств разрешённые значения могут иметь плавающую точку, например, width:50.5px.

Давным-давно getComputedStyle был создан для получения вычисленных значений, но оказалось, что окончательные значения гораздо удобнее, и стандарт изменился, так что, в настоящее время getComputedStyle фактически возвращает окончательное значение свойства, для геометрии оно обычно в пискселях.

Для правильного получения значения getComputedStyle нужно указать точное свойство. Например: paddingLeft, marginTop, borderTopWidth. При обращении к сокращённому: padding, margin, border – правильный результат не гарантируется.

Например, если есть свойства paddingLeft/paddingTop, то что мы получим вызывая getComputedStyle(elem).padding? Ничего, или, может быть, «сгенерированное» значение из известных внутренних отступов? Стандарта для этого нет.

Есть и другие несоответствия. Например, некоторые браузеры (Chrome) отображают 10px в документе ниже, а некоторые (Firefox) – нет:

<style>

body {

margin: 10px;

}

</style>

<script>

let style = getComputedStyle(document.body);

alert(style.margin); // пустая строка в Firefox

</script>

Стили, применяемые к посещённым :visited ссылкам, скрываются, посещённые ссылки могут быть окрашены с помощью псевдокласса :visited. Но getComputedStyle не даёт доступ к этой информации, чтобы произвольная страница не могла определить, посещал ли пользователь ту или иную ссылку, проверив стили.

JavaScript не видит стили, применяемые с помощью :visited. Кроме того, в CSS есть ограничение, которое запрещает в целях безопасности применять к :visited CSS-стили, изменяющие геометрию элемента. Это гарантирует, что нет обходного пути для «злой» страницы проверить, была ли ссылка посещена и, следовательно, нарушить конфиденциальность.

## События

Событие – это сигнал от браузера о том, что что-то произошло. Все DOM-узлы подают такие сигналы (хотя события бывают и не только в DOM).

Вот список самых часто используемых DOM-событий, пока просто для ознакомления:

События мыши:

* click – происходит, когда кликнули на элемент левой кнопкой мыши (на устройствах с сенсорными экранами оно происходит при касании).
* contextmenu – происходит, когда кликнули на элемент правой кнопкой мыши.
* mouseover / mouseout – когда мышь наводится на / покидает элемент.
* mousedown / mouseup – когда нажали / отжали кнопку мыши на элементе.
* mousemove – при движении мыши.

События на элементах управления:

* submit – пользователь отправил форму <form>.
* focus – пользователь фокусируется на элементе, например, нажимает на <input>.

Клавиатурные события:

* keydown и keyup – когда пользователь нажимает / отпускает клавишу.

События документа:

* DOMContentLoaded – когда HTML загружен и обработан, DOM документа полностью построен и доступен.

CSS events:

* transitionend – когда CSS-анимация завершена.

### [Обработчики событий](file:///C:\\Users\\user\\Desktop\\Web-%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D1%8B\\%D0%A7%D0%B0%D1%81%D1%82%D1%8C%202%20%D0%BC%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D1%8B\\1.5.%20%D0%92%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5%20%D0%B2%20%D0%B1%D1%80%D0%B0%D1%83%D0%B7%D0%B5%D1%80%D0%BD%D1%8B%D0%B5%20%D1%81%D0%BE%D0%B1%D1%8B%D1%82%D0%B8%D1%8F.html" \l "obrabotchiki-sobytiy)

Событию можно назначить обработчик, то есть функцию, которая сработает, как только событие произошло. Именно благодаря обработчикам JavaScript-код может реагировать на действия пользователя. Есть несколько способов назначить событию обработчик.

**[Использование атрибута HTML](file:///C:\\Users\\user\\Desktop\\Web-%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D1%8B\\%D0%A7%D0%B0%D1%81%D1%82%D1%8C%202%20%D0%BC%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D1%8B\\1.5.%20%D0%92%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5%20%D0%B2%20%D0%B1%D1%80%D0%B0%D1%83%D0%B7%D0%B5%D1%80%D0%BD%D1%8B%D0%B5%20%D1%81%D0%BE%D0%B1%D1%8B%D1%82%D0%B8%D1%8F.html" \l "ispolzovanie-atributa-html)**

Обработчик может быть назначен прямо в разметке, в атрибуте, который называется on<событие>.

Например, чтобы назначить обработчик события click на элементе input, можно использовать атрибут onclick, вот так:

<input value="Нажми меня" onclick="alert('Клик!')" type="button">

При клике мышкой на кнопке выполнится код, указанный в атрибуте onclick. Для содержимого атрибута onclick используются одинарные кавычки, так как сам атрибут находится в двойных. Если мы забудем об этом и поставим двойные кавычки внутри атрибута, вот так: onclick="alert("Click!")", код не будет работать.

Атрибут HTML-тега – не самое удобное место для написания большого количества кода, поэтому лучше создать отдельную JavaScript-функцию и вызвать её там.

Следующий пример по клику запускает функцию countRabbits():

<script>

function countRabbits() {

for(let i=1; i<=3; i++) {

alert("Кролик номер " + i);

}

}

</script>

<input type="button" onclick="countRabbits()" value="Считать кроликов!">

Как мы помним, атрибут HTML-тега не чувствителен к регистру, поэтому ONCLICK будет работать так же, как onClick и onCLICK. Но, как правило, атрибуты пишут в нижнем регистре: onclick.

**[Использование свойства DOM-объекта](file:///C:\\Users\\user\\Desktop\\Web-%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D1%8B\\%D0%A7%D0%B0%D1%81%D1%82%D1%8C%202%20%D0%BC%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D1%8B\\1.5.%20%D0%92%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5%20%D0%B2%20%D0%B1%D1%80%D0%B0%D1%83%D0%B7%D0%B5%D1%80%D0%BD%D1%8B%D0%B5%20%D1%81%D0%BE%D0%B1%D1%8B%D1%82%D0%B8%D1%8F.html" \l "ispolzovanie-svoystva-dom-obekta)**

Можно назначать обработчик, используя свойство DOM-элемента on<событие>. К примеру, elem.onclick:

<input id="elem" type="button" value="Нажми меня!">

<script>

elem.onclick = function() {

alert('Спасибо');

};

</script>

Если обработчик задан через атрибут, то браузер читает HTML-разметку, создаёт новую функцию из содержимого атрибута и записывает в свойство.

Этот способ, по сути, аналогичен предыдущему. Обработчик всегда хранится в свойстве DOM-объекта, а атрибут – лишь один из способов его инициализации. Эти два примера кода работают одинаково:

1. Только HTML:

<input type="button" onclick="alert('Клик!')" value="Кнопка">

1. HTML + JS:

<input type="button" id="button" value="Кнопка">

<script>

button.onclick = function() {

alert('Клик!');

};

</script>

Так как у элемента DOM может быть только одно свойство с именем onclick, то назначить более одного обработчика так нельзя. В примере ниже назначение через JavaScript перезапишет обработчик из атрибута:

<input type="button" id="elem" onclick="alert('Было')" value="Нажми меня">

<script>

elem.onclick = function() { // перезапишет существующий обработчик

alert('Станет'); // выведется только это

};

</script>

Кстати, обработчиком можно назначить и уже существующую функцию:

function sayThanks() {

alert('Спасибо!');

}

elem.onclick = sayThanks;

Убрать обработчик можно назначением elem.onclick = null.

**[Доступ к элементу через this](file:///C:\\Users\\user\\Desktop\\Web-%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D1%8B\\%D0%A7%D0%B0%D1%81%D1%82%D1%8C%202%20%D0%BC%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D1%8B\\1.5.%20%D0%92%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5%20%D0%B2%20%D0%B1%D1%80%D0%B0%D1%83%D0%B7%D0%B5%D1%80%D0%BD%D1%8B%D0%B5%20%D1%81%D0%BE%D0%B1%D1%8B%D1%82%D0%B8%D1%8F.html" \l "dostup-k-elementu-cherez-this)**

Внутри обработчика события this ссылается на текущий элемент, то есть на тот, на котором, как говорят, «висит» (т.е. назначен) обработчик.

В коде ниже button выводит своё содержимое, используя this.innerHTML:

<button onclick="alert(this.innerHTML)">Нажми меня</button>

### [Частые ошибки](file:///C:\\Users\\user\\Desktop\\Web-%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D1%8B\\%D0%A7%D0%B0%D1%81%D1%82%D1%8C%202%20%D0%BC%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D1%8B\\1.5.%20%D0%92%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5%20%D0%B2%20%D0%B1%D1%80%D0%B0%D1%83%D0%B7%D0%B5%D1%80%D0%BD%D1%8B%D0%B5%20%D1%81%D0%BE%D0%B1%D1%8B%D1%82%D0%B8%D1%8F.html" \l "chastye-oshibki)

Если вы только начинаете работать с событиями, обратите внимание на следующие моменты:

1. Функция должна быть присвоена как sayThanks, а не sayThanks():

// правильно

button.onclick = sayThanks;

// неправильно

button.onclick = sayThanks();

Если добавить скобки, то sayThanks() – это уже вызов функции, результат которого (равный undefined, так как функция ничего не возвращает) будет присвоен onclick, так что это не будет работать.

Но вот в разметке, в отличие от свойства, скобки нужны:

<input type="button" id="button" onclick="sayThanks()">

Это различие просто объяснить. При создании обработчика браузером из атрибута, он автоматически создаёт функцию с телом из значения атрибута: sayThanks(). Так что разметка генерирует такое свойство:

button.onclick = function() {

sayThanks(); // содержимое атрибута

};

Используйте именно функции, а не строки. Назначение обработчика строкой elem.onclick = "alert(1)" также сработает. Это сделано из соображений совместимости, но делать так не рекомендуется. Не используйте setAttribute для обработчиков. Такой вызов работать не будет:

// при нажатии на body будут ошибки,

// атрибуты всегда строки, и функция станет строкой

document.body.setAttribute('onclick', function() { alert(1) });

Регистр DOM-свойства имеет значение, используйте elem.onclick, а не elem.ONCLICK, потому что DOM-свойства чувствительны к регистру.

[addEventListener](file:///C:\\Users\\user\\Desktop\\Web-%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D1%8B\\%D0%A7%D0%B0%D1%81%D1%82%D1%8C%202%20%D0%BC%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D1%8B\\1.5.%20%D0%92%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5%20%D0%B2%20%D0%B1%D1%80%D0%B0%D1%83%D0%B7%D0%B5%D1%80%D0%BD%D1%8B%D0%B5%20%D1%81%D0%BE%D0%B1%D1%8B%D1%82%D0%B8%D1%8F.html" \l "addeventlistener)

Фундаментальный недостаток описанных выше способов назначения обработчика – невозможность повесить несколько обработчиков на одно событие. Например, одна часть кода хочет при клике на кнопку делать её подсвеченной, а другая – выдавать сообщение. Мы хотим назначить два обработчика для этого. Но новое DOM-свойство перезапишет предыдущее:

input.onclick = function() { alert(1); }

// ...

input.onclick = function() { alert(2); } // заменит предыдущий обработчик

Разработчики стандартов достаточно давно это поняли и предложили альтернативный способ назначения обработчиков при помощи специальных методов addEventListener и removeEventListener. Они свободны от указанного недостатка. Синтаксис добавления обработчика:

element.addEventListener(event, handler[, options]);

* event − имя события, например "click".
* handler − ссылка на функцию-обработчик.
* options − дополнительный объект со свойствами:
* once: если true, тогда обработчик будет автоматически удалён после выполнения.
* capture: фаза, на которой должен сработать обработчик. Так исторически сложилось, что options может быть false/true, это тоже самое, что {capture: false/true}.
* passive: если true, то указывает, что обработчик никогда не вызовет preventDefault().

Для удаления обработчика следует использовать removeEventListener:

element.removeEventListener(event, handler[, options]);

Для удаления нужно передать именно ту функцию-обработчик которая была назначена. Вот так не сработает:

elem.addEventListener( "click" , () => alert('Спасибо!'));

// ....

elem.removeEventListener( "click", () => alert('Спасибо!'));

Обработчик не будет удалён, т.к. в removeEventListener передана не та же функция, а другая, с одинаковым кодом, но это не важно. Вот так правильно:

function handler() {

alert( 'Спасибо!' );

}

input.addEventListener("click", handler);

// ....

input.removeEventListener("click", handler);

Обратим внимание – если функцию обработчик не сохранить где-либо, мы не сможем её удалить. Нет метода, который позволяет получить из элемента обработчики событий, назначенные через addEventListener.

Метод addEventListener позволяет добавлять несколько обработчиков на одно событие одного элемента, например:

<input id="elem" type="button" value="Нажми меня"/>

<script>

function handler1() {

alert('Спасибо!');

};

function handler2() {

alert('Спасибо ещё раз!');

}

elem.onclick = () => alert("Привет");

elem.addEventListener("click", handler1); // Спасибо!

elem.addEventListener("click", handler2); // Спасибо ещё раз!

</script>

Как видно из примера выше, можно одновременно назначать обработчики и через DOM-свойство и через addEventListener. Однако, во избежание путаницы, рекомендуется выбрать один способ.

Существуют события, которые нельзя назначить через DOM-свойство, но можно через addEventListener. Например, таково событие transitionend, то есть окончание CSS-анимации. Можно проверить это, запустив код в примере ниже. В большинстве браузеров, сработает лишь второй обработчик, но не первый.

<style>

input {

transition: width 1s;

width: 100px;

}

.wide {

width: 300px;

}

</style>

<input type="button" id="elem" onclick="this.classList.toggle('wide')" value="Нажми меня">

<script>

elem.ontransitionend = function() {

alert("DOM property"); // не сработает

};

elem.addEventListener("transitionend", function() {

alert("addEventListener"); // сработает по окончании анимации

});

</script>

### [Объект события](file:///C:\\Users\\user\\Desktop\\Web-%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D1%8B\\%D0%A7%D0%B0%D1%81%D1%82%D1%8C%202%20%D0%BC%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D1%8B\\1.5.%20%D0%92%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5%20%D0%B2%20%D0%B1%D1%80%D0%B0%D1%83%D0%B7%D0%B5%D1%80%D0%BD%D1%8B%D0%B5%20%D1%81%D0%BE%D0%B1%D1%8B%D1%82%D0%B8%D1%8F.html" \l "obekt-sobytiya)

Чтобы хорошо обработать событие, могут понадобиться детали того, что произошло. Не просто «клик» или «нажатие клавиши», а также – какие координаты указателя мыши, какая клавиша нажата и так далее. Когда происходит событие, браузер создаёт объект события, записывает в него детали и передаёт его в качестве аргумента функции-обработчику.

Пример ниже демонстрирует получение координат мыши из объекта события:

<input type="button" value="Нажми меня" id="elem">

<script>

elem.onclick = function(event) {

// вывести тип события, элемент и координаты клика

alert(event.type + " на " + event.currentTarget);

alert("Координаты: " + event.clientX + ":" + event.clientY);

};

</script>

Некоторые свойства объекта event:

event.type − тип события, в данном случае "click".

event.currentTarget − элемент, на котором сработал обработчик. Значение – обычно такое же, как и у this, но если обработчик является функцией-стрелкой или при помощи bind привязан другой объект в качестве this, то мы можем получить элемент из event.currentTarget.

event.clientX / event.clientY − координаты курсора в момент клика относительно окна, для событий мыши.

Есть также и ряд других свойств, в зависимости от типа событий.

При назначении обработчика в HTML, тоже можно использовать объект event, вот так:

<input type="button" onclick="alert(event.type)" value="Тип события">

Это возможно потому, что когда браузер из атрибута создаёт функцию-обработчик, то она выглядит так: function(event) { alert(event.type) }. То есть, её первый аргумент называется "event", а тело взято из атрибута.

**[Объект-обработчик: handleEvent](file:///C:\\Users\\user\\Desktop\\Web-%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81%D1%8B\\%D0%A7%D0%B0%D1%81%D1%82%D1%8C%202%20%D0%BC%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D0%B0%D0%BB%D1%8B\\1.5.%20%D0%92%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5%20%D0%B2%20%D0%B1%D1%80%D0%B0%D1%83%D0%B7%D0%B5%D1%80%D0%BD%D1%8B%D0%B5%20%D1%81%D0%BE%D0%B1%D1%8B%D1%82%D0%B8%D1%8F.html" \l "obekt-obrabotchik-handleevent)**

Мы можем назначить обработчиком не только функцию, но и объект при помощи addEventListener. В этом случае, когда происходит событие, вызывается метод объекта handleEvent, например:

<button id="elem">Нажми меня</button>

<script>

elem.addEventListener('click', {

handleEvent(event) {

alert(event.type + " на " + event.currentTarget);

}

});

</script>

Как видим, если addEventListener получает объект в качестве обработчика, он вызывает object.handleEvent(event), когда происходит событие. Мы также можем использовать класс для этого:

<button id="elem">Нажми меня</button>

<script>

class Menu {

handleEvent(event) {

switch(event.type) {

case 'mousedown':

elem.innerHTML = "Нажата кнопка мыши";

break;

case 'mouseup':

elem.innerHTML += "...и отжата.";

break;

}

}

}

let menu = new Menu();

elem.addEventListener('mousedown', menu);

elem.addEventListener('mouseup', menu);

</script>

Здесь один и тот же объект обрабатывает оба события. Обратите внимание, мы должны явно назначить оба обработчика через addEventListener. Тогда объект menu будет получать события mousedown и mouseup, но не другие (не назначенные) типы событий. Метод handleEvent не обязательно должен выполнять всю работу сам. Он может вызывать другие методы, которые заточены под обработку конкретных типов событий, вот так:

<button id="elem">Нажми меня</button>

<script>

class Menu {

handleEvent(event) {

// mousedown -> onMousedown

let method = 'on' + event.type[0].toUpperCase() + event.type.slice(1);

this[method](event);

}

onMousedown() {

elem.innerHTML = "Кнопка мыши нажата";

}

onMouseup() {

elem.innerHTML += "...и отжата.";

}

}

let menu = new Menu();

elem.addEventListener('mousedown', menu);

elem.addEventListener('mouseup', menu);

</script>

Теперь обработка событий разделена по методам, что упрощает поддержку кода.

## Вопросы для самоконтроля и задачи

1. Добавьте JavaScript к кнопке button, чтобы при нажатии элемент <div id="text"> исчезал.
2. Напишите код, выполнив задание из каждого пункта отдельной строкой:
   1. Создайте пустой объект user.
   2. Добавьте свойство name со значением John.
   3. Добавьте свойство surname со значением Smith.
   4. Измените значение свойства name на Pete.
   5. Удалите свойство name из объекта.
3. Создайте функцию clear(elem), которая удаляет всё содержимое из elem.

<ol id="elem">

<li>Привет</li>

<li>Мир</li>

</ol>

<script>

function clear(elem) { /\* ваш код \*/ }

clear(elem); // очищает список

</script>

1. Придумайте самый короткий код для проверки, пуст ли элемент elem. «Пустой» – значит нет дочерних узлов, даже текстовых.

if (/\*...ваш код проверки elem... \*/) { узел elem пуст }

1. В переменной button находится кнопка. Изначально на ней нет обработчиков. Который из обработчиков запустится? Что будет выведено при клике после выполнения кода?

button.addEventListener("click", () => alert("1"));

button.removeEventListener("click", () => alert("1"));

button.onclick = () => alert(2);

1. Создать меню, которое по нажатию открывается либо закрывается (HTML/CSS исходного документа можно и нужно менять).
2. У нас есть пустой DOM-элемент elem и строка text. Какие из этих 3-х команд работают одинаково?

elem.append(document.createTextNode(text))

elem.innerHTML = text

elem.textContent = text.

Глава 2. Библиотека jQuery



## Почему jQuery?

При всей гибкости и эффективности JavaScript, а также при всем изобилии имеющихся в этом языке встроенных функций все же сохраняется потребность в дополнительных уровнях кода, позволяющих упростить, к примеру, получение эффектов и анимации, обработку событий и применение технологии Ajax.

Более того, библиотека jQuery позволяет избежать различной интерпретации одного и того же кода JavaScript в различных браузерах, что позволяет гарантировать одинаковый внешний вид web-страниц.

Еще совсем недавнов верстальщикам приходилось писать лишний код для обеспечения совместимости с разными браузерами. До сих пор унаследованная несовместимость еще осталась, тем более, что некоторые пользователи до сих пор пользуются устаревшими браузерами. Плюс к тому, если есть желание создать что-либо не типовое, то все еще придется писать весьма существенный объем кода на JavaScript.

Для решения этих проблем было разработано множество библиотек, многие из которых, кроме этого, позволяют привязываться к объектной модели документа − DOM (Document Object Model), облегчить работу с Ajax, событиями и анимацией. В их число входят такие наиболее предпочтительные библиотеки, как AngularJS, jQuery, MooTools, Prototype, script.aculo.us и YUI (существует и множество других библиотек).

Почему рассматривается именно библиотека jQuery? Библиотека jQuery установлена более чем на 60% сайтов и востребована шире, чем все ее конкуренты [18].

Используя jQuery, вы получаете кросс-браузерную совместимость и быстрый и легкий доступ к операциям с HTML и DOM, возможность использования специальных функций для непосредственной работы с CSS, управления событиями, мощные средства для создания профессиональных эффектов и анимации, а также функции для управления обменом данными с сервером по технологии Ajax. Кроме того, jQuery является основой для широкого круга дополнительных модулей и других вспомогательных программ.

## Подключение jQuery

Есть два способа включения jQuery в web-страницы. Можно перейти на сайт jQuery, выбрать нужную версию, загрузить ее на свой сайт и пользоваться всеми ее возможностями. Или же можно воспользоваться находящейся в свободном доступе сетью доставки контента − Content Delivery Network (CDN) и просто указать ссылку на нужную версию.

Перед тем как решить, стоит ли загружать jQuery и использовать ее функции непосредственно или же воспользоваться CDN, нужно выбрать версию jQuery. В большинстве случаев выбор очевиден, поскольку вы просто отдадите предпочтение наиболее свежей версии. Но если есть намерение поддерживать работу конкретных браузеров или же вы поддерживаете устаревший сайт, работа которого зависит от определенной версии jQuery, то последний выпуск этой библиотеки может вам не подойти.

В отличие от большинства других программных средств, для использования которых вы загружаете и устанавливаете самую новую из доступных версий, jQuery совершенствовалась с учетом изменений в различных версиях браузеров и теряла обратную совместимость. В jQuery вносили различные усовершенствования, которые могли изменить работу ее новых версий на тех сайтах, которые были специально адаптированы под конкретную версию на момент ее выпуска (и под все сопутствующие ей особенности).

Разумеется, каждая более новая версия является улучшением предыдущей, и вероятность того, что вносимые усовершенствования коснутся всех базовых принципов, постоянно возрастает. Поэтому пока вы полностью не протестируете новую версию и не убедитесь в том, что операции, играющие важную роль для вашего сайта, выполняются точно так же, лучше все же продолжать использовать прежнюю версию.

Также нужно решить какую версию jQuery вам хотелось бы использовать: минимальную по размеру (сжатую), чтобы свести к минимуму требуемую полосу пропускания сети и сократить время загрузки, или несжатую (возможно, по причине того, что вам хочется вносить в нее самостоятельные правки). Как правило, наиболее удачным выбором считается минимальная по размеру версия, но большинство web-серверов поддерживают архиватор gzip, позволяющий выполнять сжатие и распаковку на лету, поэтому данный вопрос теряет свою актуальность. Также нужно учесть, что из минимизированной версии, кроме всего прочего, удалены все комментарии.

Каждая выпущенная версия jQuery может быть как в сжатой, так и в несжатой форме. Остается лишь выбрать нужную версию, щелкнуть правой кнопкой мыши на соответствующей ссылке сайта jQuery и сохранить версию на своем жестком диске. Оттуда ее можно будет загрузить на ваш web-сервер, а затем включить в web-страницу с помощью <script>-тегов примерно таким образом (для минимизированной версии выпуска 1.11.1):

<script src='http://myserver.com/jquery-1.11.1.min.js'></script>

Библиотека jQuery поддерживается несколькими сетями доставки контента (CDN). Если вы пользуетесь одной из них, то можете избавить себя от проблем, связанных с загрузкой новых версий, и использовать их, просто указав прямые ссылки на URL-адреса, поддерживаемые этими сетями. Ко всему прочему, эти сети предоставляют свои услуги совершенно бесплатно и обычно используют каналы с высокой пропускной способностью. Кроме того, CDN-сети обычно хранят свой контент в нескольких различных географических пунктах и предоставляют файл с ближайшего к вам сервера, гарантируя тем самым наиболее быструю доставку.

В общем, если вам не нужно вносить изменения в исходный код jQuery (для чего требуется его размещение на ваших собственных web-серверах) и у ваших пользователей гарантированно имеется интернет-соединение, то, скорее всего, наилучшим вариантом будет использование CDN-сетей. Тем более что пользоваться ими довольно просто, достаточно знать имя нужного файла и используемого для его загрузки корневого каталога CDN. Например, все текущие и предыдущие версии можно получить через CDN-сеть, которая используется библиотекой jQuery, с помощью следующего кода:

<script src='http://code.jquery.com/jquery-1.11.1.min.js'></script>

Основной каталог доступен по адресу http://code.jquery.com/ и за ним нужно просто дописать имя нужного для включения файла (в данном случае это jquery-1.11.1.min.js).

Библиотеку jQuery предоставляют в своих сетях как Microsoft, так и Google, поэтому для ее включения можно воспользоваться любым из следующих двух вариантов:

<script src='http://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.11.1.min.js'></script>

<script src='http://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js'></script>

В случае использования Microsoft CDN (aspnetcdn.com) в URL-адресе сначала нужно указать основной каталог ajax.aspnetcdn.com/ajax/jQuery/, а за ним – имя требуемого файла. Для Google нужно разбить имя файла (например, jquery-1.11.1.min.js) на версию (имя каталога) и имя файла таким вот образом: 1.11.1/jquery.min.js.

Еще одним преимуществом CDN-сетей является то, что вы всегда можете выбрать самую последнюю версию jQuery, следовательно, избежать необходимости ее обновления. Чтобы включить последнюю версию (выпуска из серии 1.x) из CDN-сетей jQuery или Google, нужно воспользоваться одной из двух форм <script>-тега:

<script src='http://code.jquery.com/jquery-latest.min.js'></script>

<script src='http://ajax.googleapis.com/ajax/libs/jquery/1 /jquery.min.js'></script>

При этом нужно проявить осторожность, поскольку вполне возможно, что какие-нибудь компоненты вашей web-страницы не смогут работать с будущим обновлением, и вам следует быть готовыми к тому, что какая-либо из web-страниц начнет вести себя не так, как нужно.

## Выбор элементов в jQuery

Больше всего программистов, ранее незнакомых с jQuery, удивляет символ $, который действует как фабричный метод jQuery. Он был выбран из расчета допустимости в JavaScript, краткости и отличия от имен обычной переменной, объекта или функции (метода). Им обозначается вызов функции jQuery (что также при желании можно сделать). Замысел его использования заключается в сохранении краткости и приятного внешнего вида кода, а также избавлении от излишнего кода при каждом обращении к jQuery. Кроме того, при виде этого символа другие, ранее незнакомые с вашим кодом разработчики, сразу же понимают, что в коде используется jQuery (или подобная ей библиотека).

В наипростейшем виде обращение к jQuery осуществляется набором символа $, за которым следуют заключенный в скобки селектор, точка и метод, применяемый к выбранному элементу (или элементам).

Например, для изменения семейства шрифтов всех абзацев на моноширинное можно воспользоваться следующим кодом:

$('p').css('font-family', 'monospace')

А для добавления границы к элементу <code> можно применить такой код:

$('code').css('border', '1px solid #aaa')

Взглянем на часть полноценного примера 2.1, где фрагменты, относящиеся к использованию jQuery, выделены полужирным шрифтом.

**Пример 2.1.** Простой пример применения jQuery.

<!DOCTYPE html>

<head>

<title>Первый пример jQuery</title>

<script src='jquery-1.11.1.min.js'></script>

</head>

<body>

<script>

$('code').css('border', '1px solid #aaa')

</script>

</body>

</html>

В jQuery в качестве имен функции используются либо <code>$()</code>, либо <code>jQuery()</code>. Если наряду с jQuery используются и другие библиотеки, может оказаться, что в них определена собственная $-функция. Для решения данной проблемы нужно вызвать метод noConflict:

$.noConflict()

После этого для доступа к jQuery следует вызывать функцию jQuery. Также использование символа $ можно подменить именем объекта по вашему выбору:

jq = $.noConflict()

Теперь в тех местах, где прежде применялся символ $, можно воспользоваться ключевым словом jq.

После того как вы увидели, насколько просто можно включить jQuery в web-страницу и обратиться к функциям этой библиотеки, перейдем к рассмотрению используемых в ней селекторов, которые работают точно так же, как CSS. По сути, их применение является основой работы большинства функций jQuery.

Вам остается лишь подумать о том, как бы вы оформили стиль одного или нескольких элементов с применением CSS, а затем можете использовать тот же самый селектор (или селекторы) для применения операций jQuery к этим выбранным элементам. Это означает, что вы можете воспользоваться селекторами элементов, селекторами идентификаторов, селекторами классов и любыми их сочетаниями.

### Селекторы в jQuery

Чтобы объяснить применение селекторов в jQuery, сначала рассмотрим один из более фундаментальных методов jQuery, css, с помощью которого можно динамически менять любое свойство CSS. Этому методу передаются два аргумента: имя свойства, к которому осуществляется обращение, и значение, которое к этому свойству применяется:

css('font-family', 'Arial')

Как будет показано в следующих разделах, сам по себе этот метод применять невозможно, поскольку его нужно использовать в селекторе jQuery, который выберет один или несколько элементов, чьи свойства должны быть изменены этим методом. В следующем примере содержимому всех <p>-элементов предписывается отображение с полным выравниванием по ширине:

$('p').css('text-align', 'justify')

Метод css можно также использовать для возвращения (а не для установки) вычисленного значения, для чего ему передается только имя свойства (а второй аргумент опускается). В этом случае возвращается значение первого же соответствующего селектору элемента. Например, выполнение следующего кода приведет к возвращению цвета текста того элемента, чей идентификатор (ID) имеет значение elem, и это значение будет в том же формате, в котором цвет задается при применении метода rgb:

color = $('#elem').css('color')

Следует помнить, что возвращаемое значение является вычисленным. Иными словами, jQuery будет вычислять и возвращать значение, используемое браузером на момент вызова метода, а не то исходное значение, которое могло быть присвоено свойству посредством таблицы стилей или любым другим способом. Следовательно, если текст, к примеру, показан синим цветом, значением, присвоенным переменной color в предыдущей инструкции, будет rgb(0, 0, 255), даже если цвет изначально был установлен с использованием имени цвета blue или с использованием строк шестнадцатеричных чисел #00f или #0000ff. Но это вычисленное значение всегда будет в форме, которая может быть снова назначена элементу (или любому другому элементу) при использовании в качестве второго аргумента метода css.

### **Селектор элемента**

Для выбора элемента, обрабатываемого с помощью jQuery, нужно просто указать его имя внутри круглых скобок, следующих за символом $ (или за именем функции jQuery). Например, если нужно изменить цвет фона всех элементов <blockquote>, можно воспользоваться следующей инструкцией:

$('blockquote').css('background', 'lime')

**Селектор идентификатора.** Ссылаться на элементы можно также по их идентификаторам (ID), если перед именем идентификатора поместить символ #. Следовательно, чтобы, к примеру, добавить границу к элементу с идентификатором advert, можно воспользоваться такой инструкцией:

$('#advert').css('border', '3px dashed red')

**Селектор класса.** Можно также воздействовать на группу элементов в соответствии с используемым ею классом. Например, для подчеркивания всех элементов, применяющих класс new, можно воспользоваться следующей инструкцией:

$('.new').css('text-decoration', 'underline')

**Сочетание селекторов.** Как и при использовании CSS, селекторы можно сочетать друг с другом, составляя единый jQuery-запрос, для чего, как в следующем примере, применяются запятые:

$('blockquote, #advert, .new').css('font-weight', 'bold')

В примере 2.2 все типы селекторов собраны вместе, а инструкции jQuery выделены полужирным шрифтом.

**Пример 2.2.** Использование jQuery с различными селекторами.

<!DOCTYPE html> <html>

<head>

<title>Второй пример jQuery</title>

<script src='jquery-1.11.1.min.js'></script>

</head>

<body>

<blockquote>

</blockquote>

<div id='advert'>Это реклама</div>

<p>Это мой <span class='new'>новый</span> сайт</p>

<script>

$('blockquote').css('background', 'lime')

$('#advert').css('border', '3px dashed red')

$('.new').css('text-decoration', 'underline')

$('blockquote, #advert, .new').css('font-weight', 'bold')

</script>

</body>

</html>

При всей гибкости и эффективности JavaScript, а также при всем изобилии имеющихся в этом языке встроенных функций все же сохраняется потребность в дополнительных уровнях кода, позволяющих упростить, к примеру, получение эффектов и анимации, обработку событий и применение технологии Ajax.

### Работа с выборками

Основная концепция jQuery – выбрать некоторые элементы и что-то с ними сделать. Библиотека jQuery поддерживает большинство селекторов CSS3, а также некоторые нестандартные селекторы (см. пример 2.3).

**Пример 2.3.** Различные примеры работы с выборками.

// Выбор элементов по идентификатору.

$("#myId");

// Выбор элементов по css-классу.

$(".myClass");

// Выбор элементов по значению атрибута.

$("input[name='first\_name']");

// Выбор элементов с помощью составного CSS-селектора.

$("#contents ul.people li");

// Выбор элементов с разделенным запятыми списком селекторов

$("#contents ul.people li");

// Выбор элементов с помощью псевдо-селекторов.

$("a.external:first");

$("tr:odd");

**Выбор селекторов**

Правильный выбор селекторов – один из способов улучшить производительность JavaScript. Слишком специфицированный селектор может быть малоэффективен. Селектор, такой как #myTable thead tr th.special, является избыточным, так как ту же выборку элементов сформирует более простой селектор #myTable th.special.

После того как были отобраны элементы обычно необходимо узнать, был ли найден хотя бы один элемент. Распространенной ошибкой является использование следующего кода:

// Не работает.

if ( $( "div.foo" ) ) {

...

}

Такой способ не сработает. Когда результирующая выборка формируется с помощью $(), всегда возвращается объект, который приводится к true. Даже если результат не содержит никаких элементов, код внутри оператора if все равно будет выполняться.

Лучший способ определить, есть ли какие-либо элементы, проверить свойство length, которое вернет количество выбранных элементов. Если ответ равен 0, свойство length будет иметь значение false, если используется в условном операторе:

if ( $("div.foo").length ) {

...

}

**Сохранение результата**

Библиотека JQuery не кэширует результирующую выборку. Для этого необходимо сохранить результат в переменную:

var divs = $("div");

Как только выборка сохранена в переменной, можно вызывать методы jQuery для переменной так же, как в исходной выборке. Если элементы будут добавлены на страницу позже, необходимо повторить формирование выборки или иным образом добавить их к результату, сохраненному в переменной. Сохраненные выборки не обновляются автоматически при изменении DOM.

**Уточнение и фильтрация**

Иногда выбор содержит больше элементов, чем необходимо. Библиотека jQuery предлагает несколько методов для уточнения и фильтрации выборок:

// Уточнение выборки.

$("div.foo").has("p");// div.foo элементы, которые содержат <p> теги

$("h1").not(".bar"); // h1 элементы, которые не содержат класс bar

$("ul li").filter(".current"); // элементы маркированного списка с классом current

$("ul li").first(); // только первый элемент маркированного списка

$("ul li").eq(5); // выбор шестого элемента

Некоторые методы jQuery могут использоваться для назначения или чтения некоторого значения из выборки. Когда метод вызывается со значением в качестве аргумента, он называется сеттером (setter), потому что он устанавливает (или назначает) это значение. Когда метод вызывается без аргумента, он получает (или читает) значение элемента. Такой метод называет геттером (getter). Сеттеры влияют на все элементы в выборке, тогда как геттеры возвращают запрошенное значение только для первого элемента в выборке, за исключением метода text, который извлекает значения всех элементов:

// Метод html() задает значение html-кода всех элементов h1

$("h1").html("hello world");

// Метод html()возвращает html-код первого элемента h1

$("h1").html();

### Цепочки

Если вы вызываете метод для выбора, и этот метод возвращает объект jQuery, вы можете продолжать вызывать методы jQuery для объекта, не прерывая точку с запятой. Эта практика называется «сцепление»:

$("#content")

.find("h3")

.eq(2)

.html("new text for the third h3!");

Библиотека jQuery также предоставляет метод .end(), позволяющий возратится к исходной выборке:

$("#content")

.find("h3")

.eq(2)

.html("new text for the third h3!")

.end() // возврат выборки к списку элементов с id #content

.eq(0)

.html("new text for the first h3!");

Цепочки очень удобный инструмент, который аналогичные библиотеки начали использовать после jQuery. Тем не менее, его следует использовать с осторожностью – злоупотребление цепочками может сделать код чрезвычайно сложным для изменения или отладки. Не существует правил относительно того, какой длины должна быть цепочка.

## Работа с элементами

### Основные функции

Есть много способов изменить существующий элемент. Среди наиболее распространенных задач – изменение внутреннего HTML-кода или атрибута элемента. Библиотека jQuery предлагает простые кросс-браузерные методы для такого рода манипуляций. Также можно получить информацию об элементах, используя многие из тех же методов. Вот несколько методов, которые можно использовать для получения и установки информации об элементах (см. таблицу 2.1).

**Таблица 2.1.** Основные функций для работы с элементами.

|  |  |
| --- | --- |
| Метод | Описание |
| .html() | Получить или установить содержимое HTML. |
| .text() | Получить или установить текстовое содержимое; HTML будет удален. |
| .attr() | Получить или установить значение предоставленного атрибута. |
| .width() | Получить или установить ширину в пикселях первого элемента выделения как целое число. |
| .height() | Получить или установить высоту в пикселях первого элемента выделения как целое число. |
| .position() | Получить объект с информацией о положении для первого элемента в выделении относительно его первого расположенного предка. Это только добытчик. |
| .val() | Получить или установить значение элементов формы. |

Изменение элементов тривиально, но будет влиять на все элементы, соответствующие селектору. Если необходимо изменить один элемент, обязательно укажите это в селекторе перед вызовом метода:

$("#myDiv p:first").html("New <strong>first</strong>

paragraph!");

Библиотека jQuery обладает значительным количеством методов для манипуляции атрибутами. Основные изменения вносятся с помощью метода attr, который может либо установить явное значение, либо установить возвращаемое значение функции. Когда используется синтаксис функции, она получает два аргумента: индекс элемента, атрибут которого изменяется, и текущее значение изменяемого атрибута (см. пример 2.4).

**Пример 2.4.** Работа с атрибутами.

// Присваивание одного атрибута

$("#myDiv a:first").attr("href", "newDestination.html");

// Присваивание множества атрибутов

$("#myDiv a:first" ).attr({

href: "newDestination.html",

rel: "nofollow"

});

// Использование функции для определения значения атрибута

$("#myDiv a:first").attr({

rel: "nofollow",

href: function( idx, href ) {

return "/new/" + href;

}

});

$("#myDiv a:first").attr("href", function( idx, href ) {

return "/new/" + href;

});

### Работа с CSS

Библиотека JQuery включает удобный способ получения и установки CSS-свойств элементов:

// Получение свойств

$("h1").css("fontSize"); // возвращает размер шрифта "19px"

$("h1").css("font-size"); // работает аналогично

// Установка свойств.

$("h1").css("fontSize", "100px"); // установка одного css-свойства

// установка нескольких свойств

$("h1").css({

fontSize: "100px",

color: "red"

});

Обратите внимание на стиль аргумента во второй строке – это объект, который содержит несколько свойств. Это распространенный способ передачи нескольких аргументов функции, и многие методы установки jQuery принимают объекты для установки нескольких значений одновременно.

Свойства CSS, которые обычно включают дефис, должны быть включены в JavaScript. Например, свойство CSS font-size выражается как fontSize при использовании в качестве имени свойства в JavaScript. Однако это не применяется при передаче имени свойства CSS css методу в виде строки - в этом случае будет работать форма camelCased или дефисная.

Не рекомендуется использовать метод css в качестве установщика в готовом к использованию коде, но при передаче объекта для установки CSS свойства CSS будут иметь CamelCased вместо использования дефиса.

### Классы

Иногда требуется изменить класс, используемый элементом, или просто добавить класс к элементу или удалить его из того или иного элемента. Предположим, к примеру, что у нас есть класс под названием read, который используется для придания стиля прочитанным постам блога. Добавить класс к посту можно просто с помощью метода addClass:

$('#post23').addClass('read')

За один вызов можно добавить сразу несколько классов, разделив их названия пробелами:

$('#post23').addClass('read liked')

А что делать, если пользователь решил снова пометить пост как непрочитанный, возможно, чтобы не забыть чуть позже прочитать его еще раз? В таком случае придется воспользоваться методом removeClass:

$('#post23').removeClass('read')

При этом на все остальные классы, использующиеся этим постом, не будет оказано никакого влияния.

При поддержке возможности постоянного добавления или удаления класса проще, наверное, будет воспользоваться методом toggleClass:

$('#post23').toggleClass('read')

Тогда, если пост не использовал класс, он будет добавлен, а если использовал – удален.

### Перебор элементов

Рассмотрим различные возможности библиотека jQuery для перебора элементов.

**Метод $.each**

Используя метод $.each, можно осуществить последовательный перебор элементов массивов или подобных массивам объектов путем простого прикрепления к функции, вызываемой для каждой итерации. В примере 2.5 показан массив из имен и видов домашних животных (названный pets), из которого должен быть извлечен другой массив (названный guineapigs), содержащий только имена морских свинок.

**Пример 2.5.** Вызов метода each.

<!DOCTYPE html>

<html>

<head>

<title>Использование метода each</title>

<script src='jquery-1.11.1.min.js'></script>

</head>

<body>

<div id='info'></div>

<script>

pets = {

Scratchy : 'Guinea Pig',

Squeeky : 'Guinea Pig',

Fluffy : 'Rabbit',

Thumper : 'Rabbit',

Snoopy : 'Dog',

Tiddles : 'Cat'

}

guineapigs = []

$.each(pets, function(name, type){

if (type == 'Guinea Pig') guineapigs.push(name)

})

$('#info').html('Имена морских свинок: ' + guineapigs.join(' & '))

</script>

</body>

</html>

Для выполнения поставленной задачи методу $.each передаются массив и безымянная функция для его обработки. Функция получает два аргумента, индекс в массиве (называемый name) и содержимое каждого элемента (называемое type).

Затем значение, имеющееся в type, проверяется на предмет того, не содержит ли оно строку Guinea Pig, и если содержит, значение, имеющееся в name, вставляется в массив guineapigs. По завершении содержимое массива guineapigs выводится на экран путем записи его в <div>-элемент с идентификатором info. Чтобы отделить элементы массива друг от друга, используется JavaScript-метод join с символом & в качестве разделителя. В результате загрузки примера в браузер на экране будет просто отображен текст «Имена морских свинок: Scratchy & Squeeky».

**Метод $.map**

То же самое можно сделать и при помощи метода $.map, который возвращает все значения в виде массива, освобождая от необходимости создания массива с последующим помещением в него соответствующих значений, что приходилось делать в предыдущем примере.

Вместо этого создать и наполнить массив можно, одновременно присвоив ему тот массив, который будет возвращен методом $.map (итоговый результат будет тем же, но для его получения используется меньший объем кода):

guineapigs = $.map(pets, function(type, name){

if (type == 'Guinea Pig') return name

})

### Сервисные методы

Библиотека jQuery предлагает несколько служебных методов, которые полезны для выполнения рутинных задач.

Ниже приведены примеры нескольких служебных методов:

$.trim() – удаляет начальные и конечные пробелы:

// Возвращает "lots of extra whitespace"

$.trim( " lots of extra whitespace " );

$.each() – перебирает массивы и объекты:

$.each([ "foo", "bar", "baz" ], function( idx, val ) {

console.log( "element " + idx + " is " + val );

});

$.each({ foo: "bar", baz: "bim" }, function( k, v ) {

console.log( k + " : " + v );

});

$.inArray() – возвращает индекс значения в массиве или -1, если значение отсутствует в массиве:

var myArray = [ 1, 2, 3, 5 ];

if ( $.inArray( 4, myArray ) !== -1 ) {

console.log("found it!");

}

$.extend() – изменяет свойства первого объекта, используя свойства последующих объектов:

var firstObject = { foo: "bar", a: "b" };

var secondObject = { foo: "baz" };

var newObject = $.extend( firstObject, secondObject );

console.log( firstObject.foo ); // "baz"

console.log( newObject.foo ); // "baz"

Если не требуется изменять какие-либо объекты, передаваемые в $.extend(), то передайте пустой объект в качестве первого аргумента:

var firstObject = { foo: "bar", a: "b" };

var secondObject = { foo: "baz" };

var newObject = $.extend( {}, firstObject, secondObject );

console.log( firstObject.foo ); // "bar"

console.log( newObject.foo ); // "baz"

$.proxy() – возвращает функцию, которая всегда будет выполняться в предоставленной области видимости, то есть устанавливает значение this внутри переданной функции для второго аргумента:

var myFunction = function() {

console.log( this );

};

var myObject = {

foo: "bar"

};

myFunction(); // window

var myProxyFunction = $.proxy( myFunction, myObject );

myProxyFunction(); // myObject

Если задан объект с методами, вы можете передать объект и имя метода, чтобы вернуть функцию, которая всегда будет выполняться в области объекта:

var myObject = {

myFn: function() {

console.log(this);

}

};

$("#foo").click(myObject.myFn); // HTMLElement #foo

$("#foo").click($.proxy(myObject, "myFn")); // myObject

### Хранение данных в узлах

Часто существуют данные об элементе, который необходимо хранить вместе с элементом. В JavaScript можно сделать это, добавив свойство к элементу DOM, но вам придется столкнуться с утечками памяти в некоторых браузерах. Библиотека jQuery предлагает простой способ хранения данных, связанных с элементом, и решает проблемы с памятью:

// Хранение и извлечение данных, относящихся к элементу

$("#myDiv").data("keyName", {foo: "bar"} );

$("#myDiv").data("keyName"); // возвращает {foo: "bar"}

Любые данные могут быть сохранены в элементе. Например, можно установить взаимосвязь между элементом списка и элементом <div>внутри него. Это отношение может быть установлено каждый раз при касании элемента списка, но лучшим решением будет сделать это один раз, а затем сохранить указатель <div> на элемент списка с помощью data:

// Хранение взаимосвязи между элементами с использованием.data()

$("#myList li").each(function() {

var li = $( this );

var div = li.find( "div.content" );

li.data( "contentDiv", div );

});

// Не требуется искать div снова

var firstLi = $("#myList li:first");

firstLi.data("contentDiv").html("new content");

Помимо передачи методу data одной пары ключ-значение для хранения данных, также можно передать объект, содержащий одну или несколько пар.

## Перемещение элементов DOM

### Подходы к перемещению элементов в DOM

Существует множество способов перемещения элементов по DOM, но обычно используется два подхода: перемещение выбранных элементов относительно некоторого элемента, перемещение элементов относительно выбранного элемента.

Например, jQuery предоставляет методы insertAfter и after. Метод insertAfter помещает выбранные элементы после элемента, переданного в качестве аргумента. Метод after помещает элемент, переданный в качестве аргумента после выбранного элемента. Несколько других методов работают аналогично: insertBefore и before, appendTo и append, и prependTo и prepend (см. пример 2.6).

**Пример 2.6.**  Перемещение элементов.

// Перемещение элементов с использованим различных подходов

// Перемещение первого элемента списка в конец

var li = $("#myList li:first").appendTo("#myList");

// Другой вариант:

$("#myList").append($("#myList li:first"));

Выбор нужного метода зависит от того, какие элементы выбраны, и нужно ли сохранять ссылку на элементы, которые добавляются на страницу. Если вам нужно сохранить ссылку, удобно использовать первый подход – размещение выбранных элементов относительно другого элемента, так как он возвращает размещаемые вами элементы. В этом случае необходимо выбирать из методов insertAfter, insertBefore, appendTo и prependTo.

### Копирование элементов

Такие методы, как appendTo перемещают элементы, но иногда требуется скопировать элемент. В этом случае используйте .clone():

// Скопировать первый элемент списка в конец

$("#myList li:first").clone().appendTo("#myList");

Если нужно скопировать связанные данные и события, обязательно передайте true в качестве аргумента clone.

### Удаление элементов

Есть два метода для удаления элементов со страницы: remove и detach. Используйте, remove если хотите навсегда удалить выделение со страницы. Хотя remove и возвращает удаленные элементы, к этим элементам не будут привязаны связанные с ними данные и события, если эти элементы вернуть на страницу. Используйте метод detach, если вам нужны данные и события для сохранения. Как и remove, он возвращает выборку элементов, но также сохраняет их данные и события, что позволят данные элементы на странице. Метод detach удобен для значительной модификации DOM.

Если необходимо оставить элемент на странице, но удалить его содержимое, используйте метод empty для удаления внутреннего HTML-кода элемента.

### Создание элементов

Библиотека jQuery предлагает простой способ создания новых элементов с использованием метода $():

// Создание новых элементов из HTML-кода

$("<p>This is a new paragraph</p>");

$("<li class=\"new\">new list item</li>");

// Создание нового элемента из объекта

$("<a/>", {

html: "This is a <strong>new</strong> link",

"class": "new",

href: "foo.html"

});

Обратите внимание, что атрибут class объекта указан в кавычках. Имена свойств не нужно заключать в кавычки, если они не являются зарезервированными словами.

После создания нового элемента, он не сразу добавляется на страницу. Есть несколько способов добавить элемент на страницу после его создания:

// Создание нового элемента на странице

var myNewElement = $("<p>New element</p>");

myNewElement.appendTo("#content");

// Удаление элемента p из #content

myNewElement.insertAfter("ul:last");

// Клонирование элемента p

$("ul").last().after( myNewElement.clone());

Созданный элемент не нужно хранить в переменной – есть возможность добавить элемент на страницу сразу после $(). Однако в большинстве случаев вам понадобится ссылка на добавленный элемент.

Также можно создавать элемент при добавлении его на страницу, но в этом случае нет возможности получить ссылку на вновь созданный элемент:

// Создание и добавление элемента на страницу.

$("ul").append("<li>list item</li>");

Синтаксис для добавления новых элементов на страницу прост, но необходимо учитывать, что многократное добавление элементов в DOM требует значительных ресурсов. Если требуется добавить много элементов в один и тот же контейнер, вам нужно объединить весь HTML-код в одну строку, а затем добавить эту строку в контейнер вместо поэлементного добавления. Используйте массив, чтобы собрать все части вместе, затем соедините их в одну строку для добавления:

var myItems = [];

var myList = $("#myList");

for ( var i = 0; i < 100; i++ ) {

myItems.push("<li>item " + i + "</li>");

}

myList.append(myItems.join(""));

## Обход дерева DOM

После получения первоначальный выборки элементов с помощью jQuery, вы можете получить больше информации о том, какие элементы были выбраны. Для этого необходимо их обойти. Относительно элемента можно получить родительские элементы, дочерние элементы и соседские элементы (sibling). Для этого в jQuery есть множество простых в использовании методов.

### Родители

Методы нахождения родительских элементов от выборки включают: parent, parents, parentsUntil и closest (см. пример 2.7).

**Пример 2.7.** Работа с родительскими элементами.

<div class="grandparent">

<div class="parent">

<div class="child">

<span class="subchild"></span>

</div>

</div>

<div class="surrogateParent1"></div>

<div class="surrogateParent2"></div>

</div>

// Получение родительского элемента

// возвращает [div.child]

$("span.subchild").parent();

// Получение всех родительских элементов

// возвращает[div.parent]

$("span.subchild").parents("div.parent");

// возвращает[div.child, div.parent, div.grandparent]

$("span.subchild").parents();

// Выбор всех родительских элементов до элемента, указанного в селекторе

// возвращает [div.child, div.parent]

$("span.subchild").parentsUntil("div.grandparent");

// Выбор ближайшего родителя

// возвращает [div.child]

$("span.subchild").closest("div");

// возвращает [div.child], так как он тоже включается в поиск.

$("div.child").closest("div");

### Дети

Методы поиска дочерних элементов из выборки включают children и find. Разница между этими методами заключается в том, как далеко в дочернюю ветвь делается выбор. Метод children работает только для прямых дочерних элементов, а find может рекурсивно проходить через потомков:

// Выбор непосредственных дочерних элементов

// возвращает [div.parent, div.surrogateParent1, div.surrogateParent2]

$("div.grandparent").children("div");

// Рекурсивный поиск всех дочерних элементов согласно селектору:

// возвращает [div.child, div.parent, div.surrogateParent1, div.surrogateParent2]

$("div.grandparent").find("div");

### Родственники

Все остальные методы обхода в jQuery имеют дело с поиском родственных элементов. Существует несколько основных методов, касающихся направления обхода. Можно найти предыдущие элементы с prev, следующие элементы с next, и оба с siblings. Есть также несколько других методов, которые основываются на этих основных методов: nextAll, nextUntil, prevAll и prevUntil (см. пример 2.8).

**Пример 2.8.** Работа с родственными элементами.

// Получение следующего родственного элемента

// возвращает [div.surrogateParent1]

$("div.parent").next();

// Получение предыдущего родственного элемента

// возвращает [], так как div.parent нет родственников

$("div.parent").prev();

// Получает всех следующих родственников

// возвращает [div.surrogateParent1, div.surrogateParent2]

$("div.parent").nextAll();

// возвращает [div.surrogateParent1]

$("div.parent").nextAll().first();

// возвращает [div.surrogateParent2]

$("div.parent").nextAll().last();

// Получение всех предыдущих родственников

// возвращает [div.surrogateParent1, div.parent]

$("div.surrogateParent2").prevAll();

// возвращает [div.surrogateParent1]

$("div.surrogateParent2").prevAll().first();

// возвращает [div.parent]

$("div.surrogateParent2").prevAll().last();

## События

Большинство событий инициируется действиями пользователя: при прохождении указателя мыши над элементом, щелчке кнопкой мыши или по нажатию клавиши. Но существуют и другие события, которые могут инициироваться, к примеру, по завершении загрузки документа.

Прикрепить собственный код к этим событиям с помощью jQuery не составит труда, причем сделано это будет безопасным способом, не блокирующим для другого кода получение такого же доступа к этим событиям. Вот, к примеру, как можно реализовать обработку клика на элементе:

$('#clickme').click(function(){

$('#result').html('You clicked the button!')

})

Когда будет сделан щелчок на элементе с идентификатором clickme, свойство innerHTML элемента со значением ID, равным result, будет обновлено с использованием jQuery-функции html. Полный код примера представлен в примере 2.9.

**Пример 2.9.** Обработка события.

<!DOCTYPE html> <html>

<head>

<title>События jQuery</title>

<script src='jquery-1.11.1.min.js'></script>

</head>

<body>

<button id='clickme'>Нажми меня</button>

<p id='result'>Я - абзац</p>

<script>

$('#clickme').click(function(){

$('#result').html('Вы щелкнули на кнопке!')

})

</script>

</body>

</html>

### Ожидание готовности документа

Прежде чем воздействовать на какие-либо элементы страницы, придется дождаться ее загрузки. Без jQuery это может быть выполнено с помощью события onload, но есть более эффективный кросс-браузерный jQuery-метод под названием ready, который можно вызвать для включения в работу в самый ранний из возможных моментов времени, даже раньше, чем наступит событие onload. Это означает, что jQuery может начать работать на странице намного быстрее и с минимальными задержками для пользователя.

Чтобы воспользоваться этой возможностью, поместите код jQuery внутрь следующей структуры:

$('document').ready(function(){

// Сюда нужно поместить ваш код

})

Теперь код будет ждать готовности документа и только после этого будет вызван методом ready. На самом деле можно набрать еще меньший объем кода и воспользоваться более краткой версией, показанной в примере 2.10.

**Пример 2.10.** Наименьший по объему код охватывающей функции, запускаемой по готовности документа.

$(function(){

// Сюда нужно поместить ваш код

})

Если выработать привычку помещения своих jQuery-инструкций в одну из этих двух структур, то не придется сталкиваться с тем типом ошибок, которые могут выдаваться при попытке слишком раннего обращения к DOM.

Единственная ситуация, при которой размещение сценариев в самом конце страницы может вызвать проблемы связана с тем, что документ выглядит как готовый к работе, но фактически он к ней еще не готов, или с тем, что все внешние таблицы стилей еще не загружены (реально определить это можно только тестированием), что вводит пользователей в заблуждение относительно возможности работы с документом до того, как к этому будет готов ваш сценарий. В таких случаях применяйте функцию ready, и все будет в порядке. Если же у вас есть сомнения, поместите свой сценарий в конец страницы и воспользуйтесь функцией ready, и тогда возьмете все самое лучшее от обоих вариантов.

Виды событий

Основные виды событий, используемые в библиотеке jQuery, представлены в таблице 2.2.

**Таблица 2.2.** Основные виды событий.

|  |  |  |  |
| --- | --- | --- | --- |
| № | Название | Описание | Пример |
|  | blur | инициируется, когда фокус убирается с элемента | $('input') .blur(function() { $(this).css('background', '#aaa') } ) |
|  | focus | инициируется, когда фокус попадает на элемент | $('input').focus(function() { $(this).css('background', '#ff0') } |
|  | click | инициируется от клика мышью на элемент | $('.myclass') .click( function() { $(this).slideUp() }) |
|  | dblclick | инициируется от двойного клика мышью на элемент | $('.myclass').dblclick( function() { $(this).hide() }) |
|  | keypress | инициируется от нажатия клавиши на клавиатуре | $(document).keypress(function(event)  {console.log(event.which);} |
|  | mousemove | инициируется по движению мыши | $('#pad').mousemove(function(event)  {  …  }) |
|  | mouseover | инициируется, когда курсор мыши оказывает над элементом | $('#test').mouseover(function() { $(this).html('Cut it out!') } ) |
|  | mouseout | инициируется, когда курсор мыши уходит с элемента | $('#test') .mouseout(function() { $(this).html('Try it this time...') } ) |
|  | mouseenter | инициируется, когда курсор мыши проходит над элементом | $('#test').mouseenter(function() { $(this).html('Stop!') } |
|  | mouseleave | инициируется, когда курсор мыши выходит за границу элемента | $('#test').mouseleave(function() { $(this).html('Where are you?') } ) |
|  | submit | инициируется перед отправкой формы | $('#form').submit(function(){  alert('Пожалуйста, введите имя и фамилию');  }) |

### This при обработке событий

Пример 2.11 служит иллюстрацией применения ключевого слова this.

**Пример 2.11.** Использование событий focus и blur.

<!DOCTYPE html> <html>

<head>

<title>События: blur</title>

<script src='jquery-1.11.1.min.js'></script>

</head>

<body>

<h2>Щелкните в пределах и за пределами этих полей</h2>

<input id='first'> <input><input> <input>

<script>

$('#first').focus()

$('input').focus(function(){

$(this).css('background', '#ff0')

})

$('input').blur(function(){

$(this).css('background', '#aaa')

})

</script>

</body>

</html>

При вызове события объекту this передается элемент, в отношении которого это событие было инициировано, и теперь этот объект может быть передан методу $ для обработки. Или же, поскольку он является стандартным объектом JavaScript (а не объектом jQuery), он может быть использован в качестве такого объекта. Поэтому, если хотите, можете заменить следующий код:

$(this).css('background', '#ff0')

вот этим кодом:

this.style.background = '#ff0'

## Эффекты

Библиотека jQuery позволяет создавать различные эффекты. Для этого можно воспользоваться переходами CSS3, но их динамическое управление из JavaScript не будет настолько же простым, а с jQuery все сведется к простому выбору одного или нескольких элементов с последующим применением к ним одного или нескольких эффектов.

Основными доступными эффектами являются: исчезновение, появление, постепенное проявление, растворение, скольжение различных элементов, а также их анимация. Такие эффекты могут использоваться по одному, вместе согласованно по времени или друг за другом. Поддерживаются также обратные вызовы, представляющие собой функции или методы, вызываемые только один раз по завершении операции.

### Синтаксис

Общий синтаксис при использовании эффектов следующий:

Метод (продолжительность, скорость, function(){})

При использовании эффектов следует иметь ввиду следующее:

1. Когда нет аргументов, метод вызывается немедленно и не попадает в очередь анимации.
2. Продолжительность (Duration). Когда предоставляется это значение, эффект будет наблюдаться в течение назначенного времени, которое может быть задано в миллисекундах или же строками fast (быстро) или slow (медленно).
3. Скорость выполнения эффекта (Easing). В библиотеке jQuery только два варианта изменения скорости: swing (с ускорением) и linear (линейное изменение). По умолчанию используется вариант swing, который задает более естественное изменение скорости эффекта, чем linear. Дополнительные варианты изменения скорости выполнения эффекта можно найти в таких дополнительных модулях, как jQuery UI.
4. Функция обратного вызова (Callback). Если предоставлена функция обратного вызова, она будет вызвана сразу же после завершения работы метода создания эффекта.

К примеру, метод hide можно вызвать несколькими способами:

$('#object').hide()

$('#object').hide(1000)

$('#object').hide('fast')

$('#object').hide('linear')

$('#object').hide('slow', 'linear')

$('#object').hide(myfunction)

$('#object').hide(333, myfunction)

$('#object').hide(200, 'linear', function(){

alert('Finished!')

})

Можно прикрепить вызовы функций, которые поставляют аргументы, друг к другу, а затем они будут анимированы в порядке очереди, как в следующем примере, где элемент сначала исчезнет, а затем появится снова:

$('#object').hide(1000).show(1000)

### Основные эффекты

Основные виды событий, используемые в библиотеке jQuery, представлены в таблице 2.3.

**Таблица 2.3.** Основные виды событий.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Название | Описание | Пример |
| 1 | hide | исчезновение элемента | $('#text').hide('slow', 'linear') |
| 2 | show | появление элемента | $('#show').click(function() {  $('#text').show('slow', 'linear') }) |
| 3 | toggle | переключение состояний элемента hide и show | $('#toggle').click(function() {  $('#text').toggle('slow', 'linear') }) |
| 4 | fadeOut | сокрытие элемента через растворение | $('#fadeout').click(function(){  $('#text').fadeOut( 'slow' ) }) |
| 5 | fadeIn | появление элемента через проявление | $('#fadein').click(function() { $('#text').fadeIn( 'slow' ) }) |
| 6 | fadeToggle | переключение состояний элемента fadeOut и fadeIn | $('#fadetoggle').click(function() { $('#text').fadeToggle('slow' ) }) |
| 7 | fadeTo | элемент растворяется на указанное количество процентов | $('#fadeto').click(function() {  $('#text').fadeTo( 'slow', 0.5) }) |
| 8 | slideUp | скрывает элемент скользящим движением | $('#slideup').click(function(){  $('#para').slideUp( 'slow') }) |
| 9 | slideDown | отображает элемент скользящим движением | $('#slidedown').click(function(){  $('#para').slideDown( 'slow') }) |
| 10 | slideToggle | переключение состояний элемента slideUp и slideDown | $('#slidetoggle').click(function() { $('#para').slideToggle('slow') }) |
| 11 | animate | изменяет css-свойства указанного элемента | $('#ball').animate( {  left:'270px',  top :'380px'  }, 'slow', 'linear')} |
| 12 | finish | останавливает текущую запущенную анимацию и удалит всю анимацию, выстроенную в очередь | $('#ball').click(function() {  $(this).finish() }) |

## AJAX

Технология Ajax, реализованная в библиотеке jQuery, позволяет осуществлять получение данных с сервера без перерисовки текущей страницы.

**Использование метода Post**

В примере 2.12 реализована загрузка страницы сайта в <div>-элемент, что требует гораздо меньше кода, чем при использовании чистого JavaScript, так как в нем используется один-единственный вызов метода $.post, которому передаются следующие три аргумента: URL-адрес PHP-программы на сервере, к которой нужно обратиться; данные для передачи по этому URL-адресу; безымянная функция для обработки возвращаемых данных.

**Пример 2.12.** Отправка Post-запроса по технологии Ajax.

<!DOCTYPE html>

<html> <!-- jqueryajaxpost.htm -->

<head>

<title>jQuery Ajax Post</title>

<script src='jquery-1.11.1.min.js'></script>

</head>

<body style='text-align:center'>

<h1>Загрузка web-страницы в DIV</h1>

<div id='info'>Это предложение будет заменено</div>

<script>

$.post('urlpost.php', { url : 'http://google.com' }, function(data) {

$('#info').html(data)

} )

</script>

</body>

</html>

**Использование метода Get**

Обмен данными по технологии Ajax с использованием Get-метода осуществляется практически так же легко и требует только двух следующих аргументов (см. пример 2.13): URL-адрес PHP-программы на сервер, к которой нужно обратиться (включающий строку запроса, содержащую передаваемые этой программе данные); безымянная функция для обработки возвращаемых данных.

**Пример 2.13.** Отправка Get-запроса по технологии Ajax.

<!DOCTYPE html>

<html> <!-- jqueryajaxget.htm -->

<head>

<title>jQuery Ajax Get</title>

<script src='jquery-1.11.1.min.js'></script>

</head>

<body style='text-align:center'>

<h1>Загрузка web-страницы в DIV</h1>

572 Глава 21. Введение в jQuery

<div id='info'>Это предложение будет заменено</div>

<script>

$.get('urlget.php?url=amazon.com/gp/aw', function(data){

$('#info').html(data)

} )

</script>

</body>

## Компоненты

В данной главе были рассмотрены только основные библиотеки jQuery, но она предоставляет еще значительно количество других модулей (компонентов).

В первую очередь следует упомянуть библиотеку дополнительных модулей пользовательского интерфейса jQuery (известную как jQuery UI). Применяя эти дополнения, можно добавлять к своим web-страницам методы перетаскивания, изменения размеров и сортировки, а также использовать на них еще больше эффектов, в том числе анимационных, применяя их также к цветовым переходам, получать расширенные возможности изменения скорости выполнения эффектов, а также применять пакеты виджетов для создания меню и других визуальных объектов, таких как разъезжающиеся вкладки, кнопки, календари для выбора дат, индикаторы хода выполнения, ползунки, пошаговые переключатели, вкладки, подсказки и многое другое.

У вас также есть возможность добавления jQuery широкого разнообразия бесплатных и уже готовых к работе дополнительных модулей (плагинов) от многих разработчиков. Некоторые дополнительные модули включают в себя расширенную обработку и проверку форм, демонстрацию слайдов, живые отклики на действия пользователей, работу с изображениями, еще больше эффектов анимации и многое другое.

Если разработка ведется для браузеров мобильных устройств, то необходимо рассмотреть jQuery Mobile, предназначенный для работы на этих устройствах, который представляет собой скорее среду, чем библиотеку, и предлагает специализированные, оптимизированные для использования сенсорных экранов способы навигации с применением широкого диапазона различных типов мобильного оборудования и программных средств с целью обеспечения наиболее выгодного восприятия пользователями.

Вопросы для самоконтроля и задачи

1. В чем можно писать код JavaScript?
2. Какие возможности JavaScript вы знаете?
3. Что такое AJAX?
4. Как выглядит комментарий в JavaScript?
5. Что представляет из себя библиотека jQuery?
6. Как подключить библиотеку jQuery?
7. Какие полезные jQuery-эффекты вы знаете?
8. Какие типы данных JavaScript вы знаете?
9. Для чего предназначены Свойства firstChild и lastChild?
10. Что делает Функция alert?

Глава 3. Стандарт HTML 5



## **Что такое HTML 5?**

Ключевая идея языка HTML – организация содержимого с помощью элементов, не претерпела никаких изменений с самых ранних времен Всемирной паутины. Более того, даже очень старые web-страницы без проблем обрабатываются в наиболее современных браузерах (включая и те, которые не существовали на момент создания этих страниц, например, Firefox или Chrome). Что входит в состав HTML 5?

Каждый браузер поддерживает постепенно расширяющееся подмножество возможностей HTML5. Этот подход является как хорошим, так и плохим. Хорош он потому, что браузеры могут быстро реализовать готовые части стандарта HTML5, пока другие возможности продолжают развиваться. А плох он тем, что заставляет web-разработчиков беспокоиться о проверке, поддерживает ли конкретный браузер каждую функциональную возможность, которую они хотят использовать.

Короткое описание основных категорий функциональных возможностей, охватываемых HTML5:

**Ядро HTML5.** Эта часть HTML5 составляет официальную версию спецификации организации W3C. Она содержит новые семантические элементы, новые и усовершенствованные элементы управления для web-форм, поддержку аудио и видео, а также холст для рисования с помощью JavaScript. В эту категорию входит большинство функциональных возможностей, которые наилучшим образом поддерживаются браузерами.

**Ранние возможности HTML5.** Это возможности, которые были реализованы в первоначальной спецификации HTML5, подготовленной группой WHATWG. Большинство из них − это спецификации для возможностей, требующих JavaScript и поддерживающих развитые web-приложения. Наиболее важными являются локальное хранение данных, приложения, работающие в автономном режиме, и обмен сообщениями.

**Более поздние возможности HTML5.** Это возможности следующего поколения, которые часто считаются частью HTML5, хотя они никогда не входили в стандарт HTML5. Эта категория включает спецификацию CSS3 и геолокацию.

В результате перехода поддержки HTML сначала от организации W3C к группе WHATWG, а потом обратно, возникла довольно необычная ситуация. Технически организация W3C отвечает за определение, что является официальным HTML5, а что − нет. Но в то же самое время группа WHATWG продолжает свою работу, придумывая будущие возможности HTML. Только теперь она называет его не HTML5, а просто HTML, объясняя это тем, что HTML будет продолжать существовать, как живой язык.

Для HTML-страниц пятой версии никогда не потребуется номер версии (даже в блоке указания типа документа <doctype>), а web-разработчикам никогда не понадобится обновлять свою разметку от одной версии языка к другой, чтобы она работала на новых браузерах.

## Новые элементы

В первой строке каждого HTML5-документа всегда дается описание типа документа. Это описание ясно указывает, что далее следует HTML5-содержимое, и выглядит следующим образом:

doctype <!DOCTYPE HTML>

Информация по семантическим элементам для работы со структурой страниц, семантическим элементам для работы с текстом, элементам для работы с web-формами и интерактивностью, элементами для поддержки аудио, видео и подключаемых модулей, элементами поддержки векторной графики и поддержки иных языков находится в таблице 3.1.

**Таблица 3.1.** Новые элементы HTML 5

| Категория | Элементы | Назначение |
| --- | --- | --- |
| Семантические элементы для работы со структурой страниц | <article> | Вставка статьи |
| <aside> | Вставить боковую врезку |
| <figcaption> | Описание фигуры |
| <figure> | Фигура, например рисунок с подписью |
| <footer> | Подвал |
| <header> | Шапка |
| <hgroup> | Группа заголовков |
| <nav> | Блок навигации |
| <section> | Секция |
| <details> | Блок уточнений |
| <summary> | Краткое описание |
| Семантические элементы для работы с текстом | <mark> | Выделение текста |
| <time> | Разметка времени |
| <wbr> | Место переноса текста |
| Элементы для работы с web-формами и интерактивности | <input> | Вставка других элементов |
| <datalist> | Список с вариантами выбора |
| <keygen> | Генерация пары ключей – закрытого и открытого |
| <meter> | Вывод значений в некотором диапазоне |
| <progress> | Индикатор прогресса |
| <command> | Cоздает команду в виде переключателя, флажка или обычной кнопки |
| <menu> | Предназначен для отображения списка пунктов меню |
| <output> | Определяет область в которую выводится информация |
| <label> | Устанавливает связь между определенной меткой, в качестве которой обычно выступает текст, и элементом формы |
| Элементы для поддержки аудио, видео и подключаемых модулей | <audio> | Вставка, воспроизведение аудио |
| <video> | Вставка, воспроизведение видео |
| <source> | Вставляет звуковой или видеофайл для тегов <audio> и <video> |
| <embed> | Используется для загрузки и отображения объектов, которые браузер не понимает. |
| Поддержка графики | <canvas> | Универсальный холст |
| Поддержка иных языков | <bdo> | Устанавливает направление вывода текста и преимущественно предназначен для использования с языками, где чтение происходит справа налево |
| <rp> | Используется для вывода текста в браузерах, которые не поддерживают тег <ruby>. В остальных браузерах текст, заключенный в контейнер <rp> не отображается. Тег <rp> должен идти до или после тега <rt>. |
| <rt> | Добавляет аннотацию сверху или снизу от текста, заключенного в контейнер <ruby> |
| <ruby> | Предназначен для добавления небольшой аннотации сверху или снизу от заданного текста |

Новые графические элементы <progress> и <meter> внешне похожи друг на друга, но имеют разные назначения. Элемент <progress> отображает ход выполнения задания посредством зеленой пульсирующей полоски на сером фоне. Визуально элемент <progress> похож на индикаторы выполнения, с которыми вам, вероятно, приходилось не раз встречаться, хотя его точный внешний вид зависит от браузера, в котором просматривается страница.

Элемент <meter> указывает значение в диапазоне известных значений. Внешне похож на элемент <progress>, но зеленая полоска имеет другой оттенок и не пульсирует. В зависимости от браузера цвет индикаторной полоски элемента <meter> может меняться при отображении значения, определенного как "слишком низкое" или "слишком высокое". Например, в браузере Chrome при отображении последнего значения цвет индикатора меняется с зеленого на желтый. Но самая важная разница между элементами <progress> и <meter> заключается в их семантическом значении.

Для помещения в форму элементов <meter> и <progress> нет надобности. Более того, они даже не являются настоящими элементами управления формы, т.к. не собирают информацию от посетителя web-страницы. Но официальный стандарт HTML5 включает их потому, что в некоторых отношениях они чувствуются как элементы формы, возможно, из-за графического отображения данных.

Применение элементов <progress> и <meter> не составляет никакого труда. Сначала рассмотрим элемент <progress>. Он использует атрибут value, который обозначает ход выполнения задания в виде дробной величины от 0 до 1. Графически это отображается соответствующей шириной полоски индикатора. Например, чтобы показать, что задание выполнено на 25%, атрибуту value присваивается значение 0,25.

Как альтернативу можно использовать атрибут max, чтобы установить максимальное значение и изменить масштаб индикатора. Например, при значении max, равном 200, значение value должно быть между 0 и 200. Если сделать значение value равным 50, то получим те же самые 25% заполнения индикатора:

<progress value="50" max="200"></progress>

Элемент <meter> также достаточно интеллектуальный, чтобы указывать значения выше или ниже допустимых и в то же самое время отображать их должным образом. Для этого в нем используются атрибуты low и high.

Например, значение value, превышающее значение high, но меньше значения max, будет выше какого-то требуемого значения, но все еще в пределах допустимого максимального значения. Подобным образом значение value ниже значения low, но выше значения min, не удовлетворяет какому-либо требованию низкого значения, но будет все еще в пределах допустимого минимального значения.

В HTML5 были введены следующие атрибуты тегов форм:

required – обязательность заполнения данного элемента;

disabled – отключить поле формы;

autofocus – стартовый элемент формы;

placeholder – текст-заполнитель;

Кроме того, в HTML5 появились новые элементы форм:

<input type="search"> − поле поиска;

<input type="tel"> − поле ввода телефонного номера;

<input type="email"> − поле ввода адреса электронной почты;

<input type="color"> − поле выбора цвета (рис. 3.1).

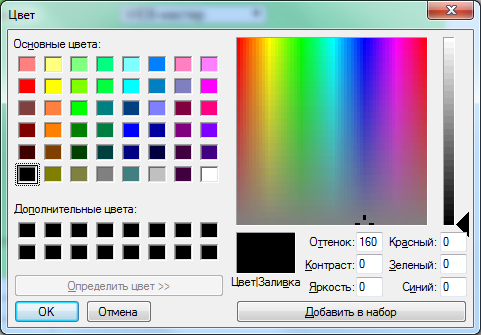


Рис. 3.1. Виджет выбора цвета.

<input type="number" min="-10" max="10"> − виджет счетчика:



Рис. 3.2. Виджет счетчика.

<input type="date"> − поле ввода даты:

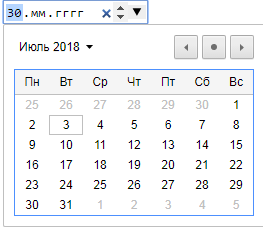


Рис. 3.3. Виджет выбора даты.

В HTML5 определяется несколько типов данных, связанных со временем. Браузеры, которые поддерживают типы дат, могут выводить удобный выпадающий календарь, в котором пользователь может выбрать требуемую дату и/или время. Это не только устраняет неопределенность относительно правильного формата даты, но также запрещает случайно (или нарочно) установить несуществующую дату. Интеллектуальные браузеры могут делать еще больше, например, поддерживать интеграцию с персональным календарем.

В таблице 3.2 ниже перечислены шесть новых форматов HTML5 для дат и времени, дано их краткое описание:

**Таблица 3.2.** Новые форматы HTML 5 для дат и времени.

|  |  |
| --- | --- |
| Тип данных | Описание |
| date | Дата по шаблону ГГГГ-ММ-ДД |
| time | Время в 24-часовом формате с необязательными секундами, по шаблону чч:мм:сс.сс |
| datetime-local | Дата и время, разделенные прописной английской буквой T (по шаблону ГГГГ-ММ-ДДTчч:мм:сс) |
| datetime | Дата и время, как и для типа datetime-local, но со смещением временного пояса. Используется такой же шаблон (ГГГГ-ММ-ДДTчч:мм:сс-чч:мм), как и для элемента <time> |
| month | Год и номер месяца по шаблону ГГГГ-ММ |
| week | Год и номер недели по шаблону ГГГГ-Номер недели |

<input type="range" min="1" max="10" value="1"> − поле ввода в виде ползункового регулятора

Элемент label необходим для сопоставления текстового описания с соответствующим полем формы. Каждый элемент label в явной или неявной форме может быть соотнесен с одним элементом формы. Неявная ассоциация:

<ul>

<li><label><input type="checkbox" name="subj" value="алгебра">алгебра</label></li>

<li><label><input type="checkbox" name="subj"

value="геометрия">геометрия</label></li>

</ul>

Явная ассоциация:

<label for="form\_username">имя пользователя:</label>

<input type="text" name="login" id="form\_username">

<label for="form\_password">пapoль :</label>

<input type="password" name="password" id="form\_password">

Элемент <datalist> − предоставляет способ присоединить выпадающий список возможных вариантов ввода к обыкновенному текстовому полю.

<input type="text" list="datl">

<datalist id="datl">

<option value="верстальщик" label="берём">верстальщик</option>

<option value="программист" label="берём">программист</option>

<option value="дизайнер" label="берём">дизайнер</option>

<option value="фигурист" label="не берем">фигурист</option>

</datalist>

</input>

Семантический элемент **<output>** предназначен для облегчения понимания разметки определенных страниц, в которых используются сценарии JavaScript. Функция этого элемента состоит просто в резервировании поля подстановки, в которое код может выводить результаты вычислений.

## **Глобальные атрибуты**

Наряду с атрибутами, характерными для конкретных тегов, в HTML5 существует и ряд атрибутов, который можно добавлять к любым тегам, поэтому входящие в эту группу атрибуты называются глобальными или универсальными.

[Accesskey](http://htmlbook.ru/html/attr/accesskey) − позволяет активировать ссылку с помощью некоторого сочетания клавиш с заданной в коде ссылки буквой или цифрой. Браузеры при этом используют различные комбинации клавиш.

[Сlass](http://htmlbook.ru/html/attr/class) − задаёт стилевой класс, который позволяет связать определенный тег со стилевым оформлением. В значении допускается указывать сразу несколько классов, разделяя их между собой пробелом.

[Contenteditable](http://htmlbook.ru/html/attr/contenteditable) − cообщает, что элемент доступен для редактирования пользователем – допускается удалять текст, и вводить новый. Также работают стандартные команды вроде отмены, вставки текста из буфера и др. Добавление этого атрибута к любому элементу позволяет редактировать содержимое этого элемента, когда оно отображается в окне браузера:

<div contentEditable>Вы можете редактировать этот текст.</div>

function startEdit() {

// Включаем редактирование элемента.

var element = document.getElementById("editableDiv");

element.contentEditable = true;

}

Обычно атрибут contentEditable в разметку не включается. Вместо этого он включается с помощью JavaScript-кода и отключается по завершению редактирования.

[Сontextmenu](http://htmlbook.ru/html/attr/contextmenu) − устанавливает контекстное меню для элемента. В качестве значения указывается идентификатор меню созданного с помощью тега <menu>.

DesignMode − похож на атрибут contentEditabie, с той разницей, что он позволяет редактировать всю web-страницу. Это может показаться слегка проблематичным, ведь при редактировании страницы отключаются события элементов. Тогда как мы сможем нажимать кнопки, чтобы управлять процессом редактирования? Решение этой проблемы простое − редактируемый документ помещается внутри элемента <iframe>, который ведет себя, как мощное окно редактирования:

<h1>Редактируемая страница</h1>

<iframe id="pageEditor" src="mypage.html" width="100%"></iframe>

<div>

<button onclick="startEdit()">Запустить редактирование</button>

<button onclick="stopEdit()">Остановить редактирование</button>

</div>

<h1>Разметка</h1>

<div id="editedHTML"></div>

[Dir](http://htmlbook.ru/html/attr/dir) − задает направление и отображение текста – слева направо или справа налево. Браузеры обычно самостоятельно различают направление текста, если он задан в кодировке Юникод, но с помощью атрибута dir можно указать, в каком направлении отображать текст. Для арабских и еврейских символов приоритетным является направление, заложенное в Юникод, поэтому на них атрибут dir действовать не будет.

[Hidden](http://htmlbook.ru/html/attr/hidden)− скрывает содержимое элемента от просмотра. Такой элемент не отображается на странице, но доступен через скрипты.

Id − задаёт идентификатор − уникальное имя элемента, которое используется для изменения его стиля и обращения к нему через скрипты. Идентификатор в коде документа должен быть в единственном экземпляре, иными словами, встречаться только один раз.

[Lang](http://htmlbook.ru/html/attr/lang)− текст документа может быть набран как на одном языке, так и содержать вставки на других языках, которые могут различаться по своим правилам оформления текста. Например, для русского, немецкого и английского языка характерны разные кавычки, в которые берется цитата. Чтобы указать язык, на котором написан текст внутри текущего элемента и применяется атрибут lang. Браузер использует его значение для правильного отображения некоторых символов.

[Spellcheck](http://htmlbook.ru/html/attr/spellcheck) − указывает браузеру проверять или нет правописание и грамматику в тексте. Хотя атрибут можно устанавливать практически для всех элементов, результат будет заметен только для полей форм (<input>, <textarea>), а также редактируемых элементов (у них установлен атрибут contenteditable).

[Style](http://htmlbook.ru/html/attr/style) − применяется для определения стиля элемента с помощью правил CSS.

T[abindex](http://htmlbook.ru/html/attr/tabindex) − атрибут tabindex устанавливает порядок получения фокуса при переходе между элементами с помощью клавиши Tab. Переход происходит от меньшего значения к большему, например, от 1 к 2, затем к 3, и так далее. При этом строгая последовательность не важна, допускается пропускать какие-то числа и начинать с любой цифры. Если значения tabindex у элементов совпадают, тогда учитывается их порядок появления в коде.

[Title](http://htmlbook.ru/html/attr/title) − cоздаёт всплывающую текстовую подсказку, которая появляется при наведении курсора на элемент. Вид такой подсказки зависит от браузера, настроек операционной системы и не может быть изменен с помощью HTML-кода или стилей.

## Преобразование xhtml в HTML 5

Форматирование страниц с применением элемента <div> и таблиц стилей − метод удобный, но не всегда очевидный для понимания. Это означает, что изучение разметки другого разработчика требует определенных усилий в том, чтобы разобраться в каждом элементе <div> и целиком во всей странице. Чтобы понять логику разметки, необходимо перескакивать туда и обратно между разметкой, таблицами стилей и отображением страницы в браузере. С таким затруднением вам придется столкнуться при рассмотрении более-менее сложной web-страницы любого разработчика, хотя, скорее всего, вы применяете такие же методы для создания своих web-страниц.

Эта ситуация заставила разработчиков задуматься, нельзя ли заменить элемент <div> чем-либо лучшим? Чем-то, что работало бы подобно <div>, но было бы более осмысленным. Чем-то, что помогло бы отделить боковые панели от заголовков, а рекламные панели от меню. Стандарт HTML5 позволяет сделать эту мечту реальностью, предоставляя набор новых элементов для структурирования страниц.

Новые семантические элементы HTML5 позволяют улучшить структуру web-страницы, добавляя смысловое значение заключенному в них содержимому. Например, новый элемент <time> обозначает действительную дату или время web-страницы. Вот самый простой пример использования этого элемента:

Регистрация начнется <time>2012-11-12</time>.

В браузере эта разметка отображается как обычный текст "Регистрация начнется 2012-11-25".

Здесь важно понимать то обстоятельство, что элемент <time> не обладает никакими встроенными возможностями форматирования. По сути, ничто не говорит устройству для чтения web-страниц о том, что дата в коде страницы заключена в дополнительный элемент. Хотя к элементу <time> можно добавить дополнительное форматирование с помощью таблицы стилей, по умолчанию он ничем не отличается от обычного текста.

В данном случае, элемент <time> предназначен содержать одну единицу информации. Но большинство семантических элементов HTML5 не такие: они служат для обозначения блоков содержимого большего размера. Например, элемент <nav> обозначает набор ссылок для перемещения по содержимому, элемент <footer> в коде обрамляет нижний колонтитул страницы и т.д.

Хотя семантические элементы наименее впечатляющие из новых возможностей HTML5, они составляют одну из самых больших групп. По сути, большинство новых элементов HTML5 является семантическими элементами.

Все семантические элементы имеют одну общую отличительную особенность: они по существу ничего не делают. В противоположность, элемент <video>, например, вставляет в web-страницу полноценный видеоплеер. Зачем же тогда утруждать себя использованием набора новых элементов, которые никак не изменяют внешний вид web-страницы? Этому есть несколько хороших причин:

1. Более удобное редактирование и сопровождение. Разметка традиционной HTML-страницы может быть трудной для понимания. Чтобы понять общую структуру и значение отдельных блоков страницы, часто приходится исследовать ее таблицу стилей. А использование семантических элементов HTML5 позволяет разработчику предоставить в разметке страницы дополнительную информацию о ее структуре. Это облегчит вам жизнь, когда потребуется редактировать эту страницу через несколько месяцев, и будет еще более кстати, если вашу работу придется редактировать кому-то другому.
2. Доступность. Одной из ключевых тем в сфере современного web-дизайна является создание доступных страниц, т. е. страниц, которые пользователи могут просматривать и перемещаться по ним с помощью программ-ридеров и вспомогательных средств.
3. Оптимизация поисковых движков. Поисковые движки наподобие Google или Yandex используют мощные поисковые роботы − автоматизированные программы, которые методически обходят Интернет и просматривают все страницы, которые они могут найти − для сканирования содержимого web-страниц и составления для них указателей в своих поисковых базах данных.  
   Чем лучше Google понимает вашу web-страницу, тем больше шансов, что он сможет сопоставить ее содержимое с поисковым запросом, и тем больше шансов, что ваша web-страница будет отображена в результатах чьего-либо поиска. В настоящее время поисковые роботы уже проверяют на наличие некоторых семантических элементов HTML5, чтобы собрать всю возможную информацию об индексирующих их web-страницах.

Из всего этого можно сделать следующий вывод: правильно используя семантические элементы, можно создавать более аккуратные и понятные web-страницы, готовые для работы со следующим поколением браузеров и web-приложений.

**Модифицирование традиционной HTML-страницы**

Легче всего начать знакомство с новыми семантическими элементами, а также обучаться их применению для структурирования страницы на классическом HTML-документе и вставить в него некоторые облагораживающие элементы HTML5. На рисунке ниже показан первый пример, который вы можете попробовать реализовать:



Рис. 3.1. Пример html-документа.

Это простая, самодостаточная web-страница, содержащая статью, хотя другие типы содержимого (например, сообщения блогов, описание продукта или короткий рассказ) вполне бы подошли для этих целей.

**Структурирование страницы старым способом**

Отформатировать страницу, показанную на рисунке, можно несколькими разными способами. К счастью, в этом примере применены самые передовые подходы HTML, поэтому в разметке нет ни намека на средства форматирования. В ней нет ни элементов для полужирного или курсивного начертания, ни вставляемых в строку разметки стилей и определенно ничего похожего на такое уродство, как устаревший элемент <font>. Вместо этого документ аккуратно отформатирован посредством связи с внешней таблицей стилей.

В следующем листинге приводится сокращенная версия разметки:

<div class="Header">

<h1>HTML 5</h1>

<p class="Teaser">Краткое описание языка разметки</p>

<p class="Byline">Информация несекретная</p>

</div>

<div class="Content">

<p><span class="LeadIn">HTML 5</span> <br>

Ключевая идея языка HTML – ...

</p>

<p>...</p>

<h2>История</h2>

<p>...</p>

<h2>Возможности</h2>

<p>...</p>

</div>

<div class="Footer">

<p class="Disclaimer">Содержимое доступно только с указанием правообладателя.</p>

<p>

<a href="#">О нас</a>

<a href="#">Контакты</a>

</p>

<p>Copyright © 2019</p>

</div>

</html>

В хорошо написанной, традиционной HTML-странице большинство работы по форматированию отдается на откуп таблице стилей посредством контейнеров <div> и <span>. Элемент <span> позволяет форматировать отрывки текста внутри другого элемента. А элемент <div> − целые блоки содержимого, и устанавливает общую структуру страницы.

В данном случае перед таблицей стилей стоит легкая задача форматирования. Для всей страницы установлена максимальная ширина в 1000 пикселов, чтобы текст не отображался длинными строками на широкоформатных мониторах. Верхний колонтитул помешен в блок синего цвета, содержимое отцентрировано:

body {

font-family: "Lucida Sans Unicode", "Lucida Grande", Verdana, sans-serif;

max-width: 1000px;

margin:0 auto;

}

/\* Форматируем элемент <div>, который представляет верхний колонтитул

(как блок с рамкой и синей заливкой) \*/

.Header {

background-color: #7695FE;

border: thin #336699 solid;

padding: 10px;

margin: 10px;

text-align: center;

}

/\* Форматируем все заголовки <hl> в элементе <div>

верхнего колонтитула (заголовок статьи) и т.д \*/

.Header h1 {

margin: 0px;

color: white;

font-size: xx-large;

}

.Header .Teaser {

margin: 0px;

font-weight: bold;

}

.Header .Byline {

font-style: italic;

font-size: small;

margin: 0px;

}

.Content {

font-size: medium;

font-family: Cambria, Cochin, Georgia, Times, "Times New Roman", serif;

padding-top: 20px;

padding-bottom: 5px;

padding-left: 50px;

padding-right: 50px;

line-height: 120%;

}

.Content .LeadIn {

font-weight: bold;

font-size: large;

font-variant: small-caps;

}

.Content h2 {

color: #24486C;

margin-bottom: 2px;

font-size: medium;

}

.Content p {

margin-top: 0px;

}

.Footer {

text-align: center;

font-size: x-small;

}

.Footer .Disclaimer {

font-style: italic;

}

.Footer p {

margin: 3px;

}

**Структурирование страницы с помощью HTML5**

Элемент <div> − основное средство современного web-дизайна. Это контейнер общего назначения, с помощью которого можно форматировать любую часть web-страницы. Недостатком элемента <div> является то, что он не предоставляет никакой информации о странице. Встретив в разметке элемент <div>, вы (или браузер, средство разработки, скрин-ридер, поисковый робот и т. п.) понимаете, что нашли отдельный блок страницы, но не знаете назначение этого блока.

Чтобы исправить такую ситуацию, в HTML5 некоторые элементы <div> можно заменить более описательными семантическими элементами. Эти семантические элементы действуют в точности таким же образом, как и элемент <div>: они группируют часть разметки в блок, но не выполняют никаких собственных операций над содержимым блока; они также предоставляют "стилевую зацепку", позволяющую присоединять форматирование. Но кроме всего этого, они также добавляют в страницу семантический смысл.

Далее приводится разметка страницы, показанной на рисунке выше, но с применением новых элементов. В частности, два элемента <div> заменены HTML5-элементами <header> и <footer>:

<header>

<h1>HTML 5</h1>

...

</header>

<div class="Content">

...

</div>

<footer>

...

</footer>

Соответственно изменятся и селекторы стилей CSS:

header {

background-color: #7695FE;

border: thin #336699 solid;

padding: 10px;

margin: 10px;

text-align: center;

}

header h1 {

/\* ... \*/

}

/\* ... \*/

footer {

text-align: center;

font-size: x-small;

}

footer .Disclaimer {

font-style: italic;

}

footer p {

margin: 3px;

}

Оба подхода одинаково действенные. Такая гибкость, без необходимости следовать жестким правилам, в HTML5 присуща многим решениям по разработке web-страниц.

Обратите внимание на блок <div>, оставшийся в содержимом. Это вполне допустимо, т.к. HTML5-страницы часто содержат смесь семантических элементов и более общего контейнера <div>. Так как в HTML5 не существует отдельного элемента для содержимого, такого как, например, <content>, то значение обычного элемента <div> остается понятным.

Наконец, есть еще один элемент, который стоит добавить в этот пример. Это элемент **<**article>, который представляет завершенную, самодостаточную единицу содержимого, например запись в блоге или новостную заметку. Элемент <article> обрамляет заголовок, строку с именем автора и основное содержимое. Добавив элемент <article> к разметке, получим следующую структуру:

<article>

<header>

<h1>Язык C#</h1>

...

</header>

<div class="Content">

...

</div>

<footer>

...

</footer>

</article>

Хотя новая разметка отображается в браузере точно так же, как и старая, сама разметка содержит довольно много дополнительной информации. Например, заглянувший на ваш сайт поисковый робот по элементу <article> может быстро разобраться, где находится содержимое страницы, а по элементу <header> − где заголовок содержимого. Элемент <footer> не будет представлять для него интереса.

В предыдущем примере мы применили элемент <header>. Но кроме этого элемента в HTML5 добавлен еще один элемент для работы с заголовками: <hgroup>. Официальные правила его применения следующие.

Прежде всего, для обычных отдельных верхних колонтитулов, не имеющих специального содержимого, вполне подойдет один из пронумерованных элементов заголовка − <h1>, <h2>, <h3> и т.д. А заголовок и его подзаголовок можно вместе обернуть в элемент <hgroup> (но в таком случае не пытайтесь втиснуть туда что-либо другое, кроме пронумерованных элементов заголовка).

Увесистый верхний колонтитул, т.е. включающий кроме заголовка еще какое-то содержимое (например, краткое содержание статьи, дату публикации, имя автора, изображение или ссылки на подтемы), можно обернуть в элемент <header>:

<header>

<hgroup>

<h1>HTML 5</h1>

<h2>Краткое описание языка разметки</h2>

</hgroup>

<p class="Byline">Информация несекретная</p>

</header>

Это, конечно же, несколько модифицированное содержимое из предыдущего примера: вместо элемента <р> подзаголовок обозначен элементом <h2>.

С первого взгляда такая разметка может показаться потенциально сбивающей с толку. С точки зрения структуры она могла бы подразумевать, что все последующее содержимое является частью подраздела, который начинается с заголовка второго уровня, что не имеет особого смысла. В конце концов, кому нужна статья, состоящая целиком из одного большого подраздела? И даже если такая структура не влияет на отображение страницы в браузере, она меняет принцип создания схемы документа браузерами и другими инструментами.

К счастью, эта проблема решается автоматически элементом <hgroup>. В структурном аспекте он обращает внимание только на заголовки верхнего уровня (в данном случае это <h1>). Другие заголовки отображаются в браузере, но они не включаются в схему документа. Такое поведение вполне логично, т. к. эти заголовки предназначены для обозначения подзаголовков, а не подразделов.

**Вставка рисунков с помощью элемента <figure>**

Многие web-страницы оформляются изображениями. Но концепт рисунка несколько иной, чем изображения. Спецификация HTML5 советует рассматривать рисунки во многом подобно рисункам в книге, иными словами, изображение, отдельное от текста, но на которое в тексте делаются ссылки.

В общем, рисунки размещаются как плавающие объекты, т.е. их вставляют в первое удобное место в тексте, вместо того чтобы закреплять возле конкретного слова или элемента. Часто рисунки снабжены подписями, которые плавают вместе с ними.

В следующем листинге показан пример разметки HTML, которая добавляет рисунок к статье с помощью стандартного синтаксиса:

<div class="FloatFigure">

<img src="image.jpg" alt="Berners-Lee" />

<p>Отец-основатель HTML</p>

</div>

В разметке подразумевается, что для размещения рисунка используется правило в таблице стилей, которое также устанавливает поля, управляет форматированием подписи к рисунку и, необязательно, создает вокруг него рамку. В следующем листинге приведен пример такого правила:

/\* Форматируем блок плавающего рисунка. \*/

.FloatFigure {

float:left;

margin-left:0px;

margin-top:0px;

margin-right:20px;

margin-bottom:0px;

}

/\* Форматирует текст подписи к рисунку \*/

.FloatFigure p {

max-width:225px;

font-size:small;

font-style:italic;

margin-bottom:5px;

}

Если вам уже приходилось раньше создавать подобный код для вставки рисунка то, возможно, будет интересно узнать, что HTML5 предоставляет новые семантические элементы, которые идеальным образом подходят для данного типа задач. В частности, вместо использования банального элемента <div> для контейнера рисунка можно использовать элемент <figure>. А текст подписи к рисунку помещается в элемент <figcaption> внутри элемента <figure> следующим образом:

<figure>

<img src="image.jpg" alt=" Berners-Lee" />

<figcaption>Отец-основатель HTML</figcaption>

</figure>

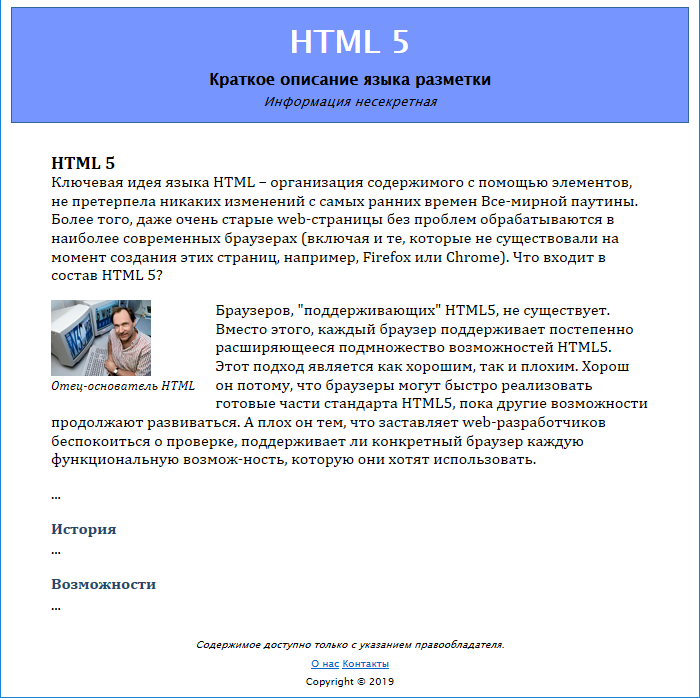


Рис. 3.2. Применение тега figure.

**Добавление боковой панели с помощью элемента <aside>**

Новый элемент <aside> представляет содержимое, которое по смыслу связано с окружающим его текстом. Например, его можно использовать как врезку в печатной работе, чтобы ввести связанную тему или развить вопрос, исследуемый в основном документе. Логично применять элемент <aside> также в случаях, когда нужно где то примостить блок объявлений, несколько ссылок на связанное содержимое или даже броскую цитату во врезке.

Этот эффект можно с легкостью создать с помощью хорошо приработанного элемента <div>, но элемент <aside> предоставляет более осмысленный способ делать разметку того же самого содержимого:

<aside class="One">

<img src="quotes\_start.png">

Большинство хороших программистов делают свою работу не потому, что ожидают оплаты или признания, а потому что получают удовольствие от программирования.

<img src="quotes\_end.png">

</aside>

В этот раз таблица стилей врезает плавающую цитату справа:

aside {

float: right;

max-width: 300px;

border-top: thin black solid;

border-bottom: thick black solid;

font-size: 20px;

line-height: 130%;

font-style: italic;

padding-top: 5px;

padding-bottom: 5px;

margin-left: 15px;

margin-bottom: 10px;

}

aside img {

width:40px;

height:30px;

vertical-align: bottom;

}

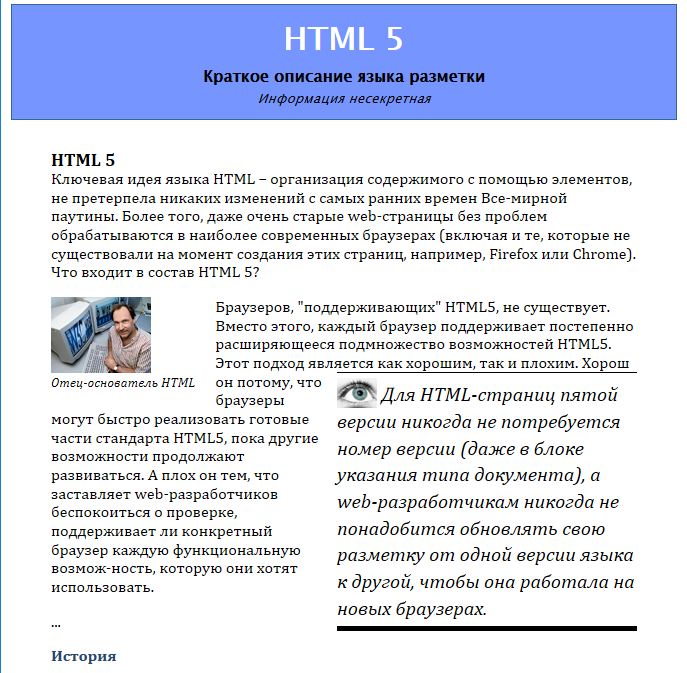


Рис. 3.3. Применение тега aside

**Создание навигационных ссылок с помощью элемента <nav>**

HTML5 позволяет создавать средства навигации по сайту с помощью семантического элемента **<nav>**. Элемент <nav> служит оболочкой для блока ссылок, указывающих на тематические разделы текущей страницы или на другие страницы web-сайта.

Большинство страниц имеет несколько блоков <nav>. Но не все ссылки требуется помещать в блоки <nav>, обычно этот блок резервируется для самых больших и наиболее важных разделов навигации страницы. Ниже показан пример использования этого элемента:

<nav>

<ul>

<li><a href="">Ссылка 1</a></li>

<li><a href="">Ссылка 2</a></li>

<li><a href="">Ссылка 3</a></li>

</ul>

</nav>

nav {

margin:10px;

max-width:1000px;

background-color:#3366CC;

text-align:center;

}

nav ul {

list-style:none;

display: inline-block;

}

nav ul li {

float:left;

margin:0px 10px 0px 10px;

}

nav ul a {

text-decoration:none;

display:block;

color:#FFFFFF;

}



Рис. 3.4. Применение тега nav

**Создание разворачиваемых блоков с помощью элементов <details> и <summary>**

Несомненно, вы часто видели на web-сайтах разворачиваемые блоки, разделы содержимого, которые можно показать или скрыть, щелкнув на заголовке. Создание разворачиваемого блока является одним из самых легких трюков web-дизайна, который можно провернуть с помощью базового сценария JavaScript. Нужно просто отреагировать на щелчок по заголовку и изменить параметры стиля, чтобы скрыть блок.

В этом отношении представляет интерес то обстоятельство, что HTML5 добавляет два семантических элемента, предназначенных для автоматизации этого поведения. Идея заключается в том, что разворачиваемый блок вставляется в элемент <details>, а заголовок блока − в элемент <summary>. Получим разметку наподобие следующей:

<details>

<summary>Раздел 1</summary>

<p>Создание разворачиваемых блоков с помощью элементов details и summary</p>

</details>

Браузеры, которые поддерживают эти элементы, показывают только заголовок и какую-либо безделицу типа небольшого треугольничка рядом с заголовком. Когда пользователь щелкает по заголовку, блок разворачивается, показывая все свое содержимое. Браузеры без поддержки элементов <details> и <summary> покажут все содержимое с самого начала, не предоставляя пользователю никакой возможности свернуть его.

Отношение web-разработчиков к элементам <details> и <summary> несколько противоречивое. Многие из них считают, что эти элементы не совсем семантические, т.е. они имеют дело больше со стилем, чем с логической структурой.

Но пока лучше всего избегать использования элементов <details> и <summary> из-за низкого уровня их поддержки браузерами. Хотя можно было бы написать обходное JavaScript-решение для браузеров, не поддерживающих этих элементов, но для этого потребуется больше усилий, чем для написания нескольких строк JavaScript для выполнения операций сворачивания/разворачивания самостоятельно и к тому же на любом браузере.

## **API интерфейсы**

API (application programming interface− интерфейс прикладного программирования) − это совокупность инструкций по программированию и стандартов, определяющих доступ к программному приложению. Это позволяет разработчику при проектировании своей продукции пользоваться мощными возможностями, предоставляемыми соответствующим API-интерфейсом.

Спецификация HTML5 предлагает несколько новых API-интерфейсов. Примеры:

* API-интерфейс двухмерного рисования, применяющийся вместе с новым тегом Canvas для отображения графиков и других визуальных материалов;
* API-интерфейс механизма кэширования для поддержки офлайновых Web-приложений;
* API-интерфейс для воспроизведения видео и аудио, применяющийся вместе с новыми тегами video и audio;
* API-интерфейс для доступа к истории просмотров и для добавления страниц к этой истории;
* API-интерфейс типа drag-and-drop для использования с атрибутом draggable;
* API-интерфейс редактирования для использования с атрибутом contenteditable;
* Хранилище на клиентской стороне с API-интерфейсами JavaScript для пар типа «ключ/значение», а также встроенные базы данных на основе SQL.

## Аудио и видео

Не прибегая к HTML5, видео в web-страницу можно добавить двумя способами. Самый простой состоит в использовании элемента <embed>. Потом браузер создаст видеоокно под проигрыватель Windows Media Player, Apple QuickTime или какой-либо другой проигрыватель и разместит его на странице.

Основная проблема с этим подходом заключается в том, что он полностью отдает разработчика на милость поддержки браузера. Разработчик не властен управлять воспроизведением, может не иметь возможности буферизировать видео, чтобы предотвратить задержки с воспроизведением, а также не знает, сможет ли данный видеофайл воспроизводиться в разных браузерах или операционных системах.

Второй подход заключается в использовании подключаемого модуля браузера, наподобие Silverlight корпорации Microsoft или Flash от компании Adobe.

До недавнего времени использование модуля Flash полностью решало задачу поддержки видеосодержимого браузерами. Но подход с использованием Flash также не идеальный. Чтобы вставить видео Flash в web-страницу, в нее нужно вставить определенный объем неконструктивной разметки, содержащей элементы <object> и <embed>. Кроме этого, видеофайлы для показа нужно закодировать в требуемый формат, а также может потребоваться приобрести дорогостоящий инструментарий Flash-разработки и научиться пользоваться им, что может потребовать серьезных усилий. Но хуже всего, это новая волна мобильных устройств компании Apple − iPhone и iPad. Эти устройства органически не воспринимают Flash и выводят пустую рамку в том месте страницы, где вставлено это видео.

Модули расширения также имеют свойство временами сбоить. Причина кроется в принципе их работы. Например, при посещении страницы, на которой используется видео Flash, браузер позволяет модулю Flash взять под контроль прямоугольную область где-нибудь на странице. В большинстве случаев этот подход работает нормально, но незначительные ошибки или нестандартные системные настройки могут вызвать неожиданные помехи и сбои, такие как, например, искаженное видео или потребление web-страницей огромных объемов памяти.

Уже в 2010 г. сервис YouTube ввел в действие HTML-видеопроигрыватель на испытательной основе. Сейчас такие проигрыватели используются повсеместно.

<**audio**> и <**video**>.

В следующем коде приведён простейший пример использования элемента <audio>:

<p>Оцените новую песню <cite>Alizee - Wake Me Up</cite></p>

<audio src="mysong.mp3" controls></audio>

Атрибут src содержит имя аудиофайла для воспроизведения, а атрибут controls указывает браузеру, что нужно отобразить базовые элементы управления воспроизведением. Своим внешним видом эти элементы управления слегка отличаются от браузера к браузеру, но все они имеют одинаковое назначение: разрешают пользователю начинать и останавливать воспроизведение, переходить в другое место записи и регулировать громкость.

Кроме атрибута controls элемент <audio> поддерживает еще три атрибута: preload, auotoplay и loop. Атрибут preload указывает браузеру способ загрузки аудиофайла. Значение auto этого атрибута указывает браузеру загружать аудиофайл полностью, чтобы он был доступен, когда пользователь нажмет кнопку воспроизведения. Конечно же, загрузка осуществляется в фоновом режиме, чтобы посетитель web-страницы мог перемещаться по странице и просматривать ее, не дожидаясь завершения загрузки аудиофайла.

Атрибут preload может принимать еще два значения. Значение metadata указывает браузеру загрузить первую небольшую часть файла, достаточную, чтобы определить некоторые его основные характеристики (например, общий размер файла). Значение none указывает браузеру не загружать аудиофайл автоматически. Эти опции можно использовать для того, чтобы сэкономить пропускную способность подключения, например, если страница содержит большое число элементов <audio>, но ожидается, что пользователь будет проигрывать лишь некоторые из них.

Когда атрибуту preload задано значение none или metadata, загрузка аудиофайла начинается после того, как пользователь нажмет кнопку воспроизведения. К счастью, браузеры могут без проблем проигрывать одну часть аудиофайла в то время, как грузится другая, если только интернет-подключение не слишком медленное.

Если значение атрибута preload не установлено, то браузеры действуют по своему индивидуальному усмотрению. Большинство браузеров предполагает auto в качестве значения по умолчанию, но в Firefox это metadata. Кроме этого, важно помнить, что атрибут preload не является обязательным для выполнения правилом, а рекомендацией браузеру желаемого действия, которую он может игнорировать зависимости от других обстоятельств. А некоторые устаревшие браузеры вообще не обращают внимания на атрибут preload.

Если вставить в разметку несколько элементов <audio>, то браузер создаст отдельную полосу элементов управления воспроизведением для каждого из них. Посетитель web-страницы может прослушивать аудио на одном из них или на всех сразу.

Атрибут autoplay указывает браузеру начать воспроизведение сразу же после завершения загрузки страницы.

Наконец, атрибут loop указывает браузеру повторять воспроизведение.

С элементом <audio> хорошо идет в паре элемент <video>. Он применяет такие же атрибуты src, controls, autoplay и loop.

Как и в случае с элементом <audio>, атрибут controls указывает браузеру создать набор элементов управления воспроизведением. В большинстве браузеров эти элементы скрываются при щелчке где-нибудь в области страницы и отображаются опять при наведении курсора мыши на область видеоплеера.

Атрибуты height и width устанавливают высоту и ширину окна воспроизведения в пикселах, соответственно.

Размеры окна воспроизведения должны совпадать с размером видео, но лучше явно указать их, чтобы оформление страницы не искажалось до того, как видеофайл загрузится (или если видеофайл вовсе не загружается).

Наконец, атрибут poster позволяет указать изображение, которое можно использоваться вместо видео:

Одним из основных поводов заняться углубленным изучением программирования элементов <audio> и <video> в JavaScript будет создание собственного проигрывателя. Основная идея заключается в простоте процесса – удаляем атрибут controls, чтобы было только окно воспроизведения, и добавляем внизу его свои кнопки управления воспроизведением. А чтобы эти кнопки функционировали, добавляем соответствующий JavaScript-код.

Любому видеопроигрывателю требуется базовый набор кнопок управления воспроизведением. Стандартные кнопки управления воспроизведением проигрывателя создаются следующим кодом:

<video id="videoPlayer" width="600" height="400">

<source src="video.mp4" type="video/mp4">

<source src="video.ogv" type="video/ogg">

</video>

<div>

<button onclick="play()">Play</button>

<button onclick="pause()">Pause</button>

<button onclick="stop()">Stop</button>

<button onclick="speedUp()">Быстрей</button>

<button onclick="slowDown()">Медленее</button>

<button onclick="normalSpeed()">Нормальная скорость</button>

</div>

JS

var video;

var display;

window.onload = function() {

video = document.getElementById("videoPlayer");

};

function play() {

video.play();

}

function pause() {

video.pause();

}

function stop() {

video.pause();

video.currentTime = 0;

}

Тег <source> позволяет указать несколько мультимедийных файлов в различном формате. Это необходимо, чтобы браузер смог выбрать подходящий / поддерживаемый формат для загрузки и дальнейшего воспроизведения аудио, либо видео. В настоящее время существует 3 поддерживаемых формата видео (MP4, WebM, Ogg) и 3 формата аудио (MP3, Wav, Ogg).

Элемент <embed> также позволяет добавлять на страницы мультимедийные файлы. Он не входил в стандарт, но широко поддерживался браузерами, пока в конце концов не обрел официальный статус в спецификации HTML5. Для некоторых мультимедийных файлов требуется использовать элемент embed, который часто применяется в качестве запасного варианта в элементе object, чтобы угодить всем браузерам. Пример его использования:

<embed src="images/examle.swf">

У тега <embed> имеется свой набор атрибутов:

width, height − высота объекта

align − определяет как объект будет выравниваться на странице и способ его обтекания текстом

hidden − указывает, скрыть объект на странице или нет

pluginspage − адрес страницы в Интернете, откуда можно скачать и установить плагин к браузеру

type − MIME-тип объекта

## Canvas

Элемент <**canvas**> предоставляет рабочее пространство для рисования. С точки зрения разметки, это простой до предела элемент с тремя атрибутами – id, width и height.

Обычно холст отображается как пустой прямоугольник без рамки, т.е. он не виден вообще. Чтобы сделать холст видимым, с помощью таблицы стилей ему можно дать цветной фон или рамку. Чтобы всё функционировало должным образом, необходимо прописать следующий код в JS:

var canvas = document.getElementById("drawingCanvas");

var context = canvas.getContext("2d");

В таблице 3.3 представлены основные методы canvas:

**Таблица 3.3.** Основные методы canvas

| Результат | Методы | Описание |
| --- | --- | --- |
| Прямые пути | moveTo | Первая команда начала построения контура |
| lineTo | Рисует отрезок через 2 точки |
| stroke | Рисование прямоугольного контура |
| closePath | Этот метод пытается закрыть фигуру, рисуя прямую линию из текущей точки в начальную |
| Фигуры | fillRect | Рисование заполненного прямоугольника. |
| strokeRect | Рисование прямоугольного контура. |
| Кривые линии | arc | Дуга через заданные точки и радиус |
| bezierCurveTo | Рисуется кубическая кривая Безье с текущей позиции пера в конечную точку с координатами x и y, используя две контрольные точки с координатами |
| quadraticCurveTo | Рисуется квадратичная кривая Безье с текущей позиции пера в конечную точку с координатами x и y, используя контрольную точку |
| Трансформации | translate | изменяет позицию точки начала координат |
| rotate | Все что отрисовывается после выполнения этого метода будет повернуто на заданный угол относительно начала координат. |
| Прозрачность | fillStyle | Позволяет задавать цвета фигурам , автоматически будет казаться , что одна наложена на другую например.Создаётся прозрачность. |
| strokeStyle | Тоже самое |
| Вставка изображений и текста | drawImage | Вставляет изображение по координатам холста. |

Задача сохранения содержимого холста требует рассмотрения множества опций. Прежде всего, нужно решить, каким образом получить данные рисунка. Для решения этой задачи холст предоставляет три возможных подхода:

1. Использовать URL данных. При этом подходе содержимое холста преобразуется в файл изображения, которое потом переводится в последовательность символов, оформленных в виде URL. Это позволяет получить аккуратный и компактный способ для перемещения данных изображения (например, его можно передать элементу <img> и отослать на web-сервер).
2. Использовать метод getImageData(). Этот подход позволяет получить "сырые" пиксельные данные, которыми можно потом манипулировать как угодно.
3. Сохранять список "шагов". Например, можно организовать массив, содержащий список всех линий, нарисованных на холсте. Эти данные можно потом сохранить и использовать для воспроизведения изображения. Данный подход требует меньше места для хранения изображения, а также предоставляет большую гибкость для последующей работы с ним.

Но это еще не все. Определившись с типом содержимого для сохранения, нужно решить, где это содержимое сохранить. Возможны, среди прочих, следующие опции:

1. В файле изображения. Например, содержимое холста сохраняется в виде файла формата PNG или JPEG на локальном жестком диске.
2. На web-сервере. После передачи данных web-серверу последний может сохранить их в файле или базе данных и предоставить при следующем посещении страницы пользователем.

Для сохранения содержимого холста в нашей программе рисования применяется возможность, которая называется URL данных (data URL). Чтобы получить URL для текущих данных, мы просто используем метод холста toDataURL(). Если вызвать метод toDataURL(), не передав ему никаких параметров, то получим изображение в формате PNG. Альтернативно, методу можно указать требуемый формат изображения:

var URL = canvas.toDataURL("image/jpeg");

Сделать анимацию из элемента на холсте HTML5 достаточно просто. Для этого устанавливается таймер, который постоянно вызывает элемент, обычно 30 или 40 раз в секунду. При каждом вызове код полностью обновляет содержимое всего холста. Если код написан правильно, постоянно сменяющиеся кадры сольются в плавную, реалистичную анимацию.

JavaScript предоставляет два способа для управления этим повторяющимся обновлением содержимого холста:

Функция setTimeout(). Эта функция дает указание браузеру подождать несколько миллисекунд, а потом исполнить фрагмент кода, в данном случае код для обновления содержимого холста. По окончании исполнения кода функция setTimeout() выполняется опять, вновь вызывая код обновления холста, и так до тех пор, пока анимацию не нужно завершить.

Функция setInterval(). Эта функция дает указание браузеру исполнять фрагмент кода через регулярный интервал времени, например каждые 20 мс. Эффект от этой функции, по большому счету, такой же, как и от функции setTimeout(), но функцию setInterval() нужно вызывать только один раз. Чтобы остановить повторяющийся вызов кода браузером, исполняется функция clearInterval().

Если код рисования быстрый и эффективный, обе функции дают одинаковый эффект. В противном случае функция setInterval() даст лучший результат, выполняя обновление точно в требуемое время, но, возможно, за счет производительности. При худшем стечении обстоятельств, когда исполнение кода рисования занимает несколько больше времени, чем установленный интервал, браузер будет напрягаться, чтобы поспевать, код рисования будет исполняться постоянно и страница может на мгновение зависать.

При вызове функции setTimeout() передаются два параметра: название функции, которою нужно выполнить, и период времени ожидания перед исполнением этой функции. Время ожидания указывается в миллисекундах (т.е. тысячных секунды), таким образом, 20 мс (обычная задержка при анимации) равняется 0,02 с.

Рассмотрим анимацию некольких объектов одновременно на примере падающих мячиков, JS код:

function drawFrame() {

// Очистить холст

context.clearRect(0, 0, canvas.width, canvas.height);

// Вызываем метод beginPath(), чтобы убедиться,

// что мы не рисуем часть уже нарисованного содержимого холста

context.beginPath();

// Перебираем все мячики

for(var i=0; i<balls.length; i++) {

// Перемещаем каждый мячик в его новую позицию

var ball = balls[i];

ball.x += ball.dx;

ball.y += ball.dy;

// Добавляем эффект "гравитации", который ускоряет падение мячика

if ((ball.y) < canvas.height) ball.dy += 0.22;

// Добавляем эффект "трения", который замедляет движение мячика

ball.dx = ball.dx \* 0.998;

// Если мячик натолкнулся на край холста, отбиваем его

if ((ball.x + ball.radius > canvas.width) || (ball.x - ball.radius < 0)) {

ball.dx = -ball.dx;

}

// Если мячик упал вниз, отбиваем его, но слегка уменьшаем скорость

if ((ball.y + ball.radius > canvas.height) || (ball.y - ball.radius < 0)) {

ball.dy = -ball.dy\*0.96;

}

// Проверяем, хочет ли пользователь соединительные линии

if (!document.getElementById("connectedBalls").checked) {

context.beginPath();

context.fillStyle = ball.fillColor;

}

else {

context.fillStyle = "white";

}

// Рисуем мячик

context.arc(ball.x, ball.y, ball.radius, 0, Math.PI\*2);

context.lineWidth = 1;

context.fill();

context.stroke();

}

// Рисуем следующий кадр через 20 миллисекунд

setTimeout("drawFrame()", 20);

}

## Web Storage и File API

### Local Storage

Объект localStorage используется для хранения данных для всего web-сайта на постоянной основе. Это означает, что, если web-страница сохранит данные в локальном хранилище, эти данные будут доступны для пользователя, когда он возвратится на эту web-страницу на следующий день, на следующей неделе или в следующем году.

Конечно же, большинство браузеров также предоставляет пользователю возможность очистить локальное хранилище. В некоторых браузерах она реализована как стратегия "все или ничего", и посредством ее удаляются все локальные данные, во многом подобно тому, как удаляются cookies-файлы (в действительности, в некоторых браузерах система cookies и локальное хранилище взаимосвязаны, так что единственным способом удалить локальные данные будет удаление cookies.) А другие браузеры могут предоставлять пользователю возможность просмотра данных для каждого отдельного web-сайта и удалять данные для выбранного сайта или сайтов.

Прежде чем поместить фрагмент информации в локальное хранилище или хранилище сеансов, ему необходимо присвоить описательное имя. Это имя называется ключом (key) и нужно для того, чтобы данные можно было извлечь в будущем.

Синтаксис для сохранения фрагмента данных следующий:

localStorage[keyName] = data;

Задача удаления хранящихся в web-хранилище данных проста до предела. Для удаления отдельного ненужного элемента используется метод removeItem(), которому передается соответствующий ключ:

// JS

localStorage.removeItem("localData");

### File API

Возможность web-хранилища поддерживается в HTML5 на хорошем уровне. Но это не единственный способ получения информации web-страницами. В эту область понемногу вводится несколько других стандартов, направленных на выполнение разных типов задач хранения данных. Одним из таких стандартов является File API, который технически не входит в HTML5, но имеет хороший уровень поддержки на всех современных браузерах.

Судя по расплывчатому названию этого стандарта, может показаться, что это всеохватывающий стандарт для чтения и записи файлов на жесткий диск клиента. Но этот стандарт не настолько амбициозный или мощный. Он просто разрешает посетителю web-сайта выбрать файл на своем жестком диске и передать его непосредственно коду JavaScript, исполняющемуся на просматриваемой web-странице. Код может открыть этот файл и работать с его данными, будто это простой текст или что-то более сложное. Здесь ключевым аспектом является то обстоятельство, что файл передается непосредственно коду JavaScript. В отличие от обычной выгрузки файла, он никогда не отправляется на web-сервер.

Также важно знать, что File API не может делать. Самое важное, что он не может изменять файлы или создавать новые файлы. Чтобы сохранить какие-либо данные, нужно прибегать к другому механизму, например данные можно отправить на web-сервер посредством запроса XMLHttpRequest или же поместить их в локальное хранилище.

Судя по этому, можно подумать, что интерфейс File API менее полезен, чем локальное хранилище, и для большинства web-сайтов это будет правильный вывод. Но этот стандарт приоткрывает дверь в область, в которую HTML раньше не входил, по крайней мере без помощи модулей расширений.

В настоящее время интерфейс File API является необходимой функциональностью для определенных типов специализированных приложений, но в будущем его возможности могут быть расширены, и важность его значительно возрастет. Например, будущие версии интерфейса могут позволять web-страницам сохранять файлы на жесткий диск клиента при условии, что пользователь контролирует имя файла и место его сохранения, используя диалоговое окно "Сохранить как". Модули расширения, наподобие Flash, уже оснащены такой способностью.

Прежде чем интерфейс File API сможет что-либо сделать с файлом, ему нужно этот файл получить. Эту задачу можно выполнить тремя разными способами, но все они одинаковые в одном ключевом аспекте − web-страница может получить файл только в том случае, если посетитель явно выберет и предоставит его web-странице.

Способы получения файла следующие:

1. Посредством элемента <input>. Присвоив атрибуту type значение file, мы получим стандартное окно для закачивания файла. Но с помощью небольшого сценария JavaScript и File API этот файл можно открыть локально.
2. Посредством скрытого элемента <input>. Элемент <input> очень непривлекательный. Чтобы не обезображивать им свою страницу, его можно скрыть и создать более прилично выглядящую кнопку. Нажатие этой кнопки активирует JavaScript-код, вызывающий метод click() скрытого элемента <input>, который открывает стандартное диалоговое окно выбора файла.
3. Посредством метода drag and drop. Если браузер поддерживает этот метод, файл можно перетащить с рабочего стола или окна браузера и отпустить его в определенной области web-страницы.

Самой простой операцией File API будет чтение содержимого простого текстового файла. Создание этого примера начинается с элемента <input type="file">, который создает текстовое поле и кнопку "Выберите файл":

Но в то время как элемент <input> обычно вставляется в контейнер <form>, чтобы файл можно было закачать на web-сервер, в данном случае наш элемент <input> играет самостоятельную роль. Когда посетитель страницы выбирает файл, активируется событие onchange элемента <input>, что в свою очередь активирует функцию processFiles(). Как раз на этом этапе и открывается файл посредством самого обыкновенного кода JavaScript.

Теперь рассмотрим по частям функцию processFiles(). Сперва нам нужно взять первый файл из коллекции файлов, предоставленных элементом <input>. Если явно не разрешить выбор нескольких файлов (посредством атрибута multiple), коллекция файлов будет гарантированно содержать только один файл, размещенный в элементе 0 массива файлов:

Все объекты файлов обладают тремя потенциально полезными свойствами. Свойство name сообщает нам имя файла (без пути), свойство size указывает размер файла в байтах, а свойство type информирует о MIME-типе файла, если его можно определить. Эти свойства можно считывать и использовать их в дополнительной логике, например, отказаться обрабатывать файлы больше определенного размера или разрешить обрабатывать файлы только определенного типа.

Далее создается объект FileReader для обработки файла. Теперь мы почти готовы вызвать один из методов объекта FileReader, чтобы извлечь содержимое файла. Но эти методы являются асинхронными. Это означает, что они начинают чтение файла, но не ожидают получения данных. Поэтому, чтобы получить данные, сначала нужно обработать событие onload. Наконец, подготовив этот обработчик события, можно вызывать метод readAsText() объекта FileReader. Этот метод сбрасывает все содержимое файла в одну длинную строку, вставляемую в свойство e.target.result, которое в свою очередь отправляется событию onload. Метод readAsText() работает должным образом только для текстового содержимого файла, но не для двоичного. Это означает, что он идеально подходит для работы с файлами HTML, как показано на рисунке выше. Кроме метода readAsText(), объект FileReader имеет еще несколько других методов для чтения файлов: readAsBinaryString(), readAsDataURL() и readAsArrayBuffer(), но последний метод не поддерживается в Firefox.

Метод readAsBinary() предоставляет web-приложению возможность считывать двоичные данные, хотя он вставляет эти данные в текстовую строку несколько неуклюже, что не является особенно эффективным. А если вы еще захотите разобраться с этими данными, то для этого вам придется мучиться с исключительно запутанным кодом.

Метод readAsDataURL() предоставляет легкий способ захватывать данные изображения. Мы рассмотрим применение этого метода далее, но сначала мы выясним, как сделать нашу страницу более красивой.

Web-разработчики сходятся во мнении: стандартный элемент управления <input>, применяемый для выгрузки файлов, выглядит далеко не лучшим образом. Но хотя нам не избежать использования его, совсем необязательно, чтобы пользователи его видели. Мы можем просто скрыть его с помощью следующего стилевого правила:

#fileInput {

display:none;

}

Теперь нам нужно добавить новый элемент управления, который будет инициировать процесс предоставления файла. Для этого достаточно обычной кнопки со ссылкой, которую мы можем разукрасить каким угодно образом:

<button onclick="showFileInput()">Добавить файл</button>

Последним шагом будет обработка нажатия кнопки путем инициализации вручную элемента <input> через вызов метода click() этого элемента:

function showFileInput() {

var fileInput = document.getElementById("fileInput");

fileInput.click();

}

Теперь нажатие этой кнопки запускает функцию showFileInput(), которая "нажимает" скрытую кнопку "Выберите файл" и отображает диалоговое окно для выбора файла. Это, в свою очередь, активирует событие onchange скрытого элемента <input>, которое запускает функцию processFiles() точно таким же образом, как и раньше.

## **Средства взаимодействия с сервером**

### XMLHttpRequest

Объект XMLHttpRequest является основной частью большинства современных web-приложений. Ключевая идея в основе объекта XMLHttpRequest состоит в том, что он позволяет коду JavaScript самостоятельно направлять запросы к серверу, когда приложению требуются дополнительные данные. Эти запросы осуществляются асинхронно, что означает, что web-страница остается доступной для работы даже в процессе выполнения такого запроса. Более того, посетитель страницы никогда даже не догадывается о выполняющемся за кулисами запросе (если только не выводится соответствующее извещение или индикатор хода выполнения).

Объект XMLHttpRequest является идеальным инструментом для получения данных с web-сервера. Далее приведено несколько примеров данных, которые можно получить с web-сервера посредством этого объекта:

Данные, хранящиеся на сервере. Это может быть информация в файле или, в большинстве случаев, в базе данных. Например, посетитель может затребовать информацию о продукте или о компании.

Данные, которые можно вычислить только на сервере. Например, web-сайт может содержать на сервере программу, выполняющую сложные вычисления. Эти вычисления можно было бы попытаться выполнить в браузере с помощью JavaScript, но по разным причинам это может оказаться невозможным.

Например, математические возможности JavaScript могут быть ниже требуемого уровня или же код может не иметь доступа к некоторым данным, требующимся для вычислений. Также код может выполнять секретные вычисления, которые вы не хотите выставлять напоказ для всех, кто пожелает их видеть, чтобы избежать потенциальных несанкционированных манипуляций с вычислениями. Или же вычисления могут требовать возможностей, которые настольный компьютер не в силах предоставить, как это может сделать мощный web-сервер (например, прорисовка трехмерной сцены). Во всех подобных случаях имеет смысл выполнять вычисления на сервере.

Данные, которые находятся на чьем-то другом сервере. Web-страница не может обращаться напрямую к чьему-то другому web-серверу. Но она может вызвать программу на своем web-сервере (посредством объекта XMLHttpRequest) которая в свою очередь может вызвать другой web-сервер, получить от него данные и возвратить эти данные на исходную страницу.

Объект XMLHttpRequest позволяет web-странице задать серверу вопрос и получить на него немедленный ответ. Но это обмен типа "один к одному" — после предоставления сервером ответа данный сеанс взаимодействия завершается. Web-сервер не может повременить несколько минут и отправить странице другое сообщение с обновленной информацией.

Но некоторые типы web-страницы могли бы извлечь пользу из более длительных сеансов взаимодействия с сервером. Возьмем, например, страницу биржевых котировок на Google Finance. Если оставить это страницу открытой, то периодически мы будем видеть автоматическое обновление котировок. Подобным образом доставляются новые сообщения в ящик поступающих сообщений любой программы электронной почты с web-интерфейсом.

Во всех этих примерах применяется метод, называющийся опросом (polling). Суть его состоит в том, что web-страница периодически (например, каждые несколько минут) запрашивает у web-сервера новые данные. Для реализации этого механизма обновления web-страницы используется функция JavaScript setInterval() или setTimeout(), которая активирует код по истечении установленного периода времени.

### Серверные события

Метод опроса является разумным решением, но иногда неэффективным. Во многих случаях это означает, что нужно вызывать web-сервер и устанавливать новое подключение только для того, чтобы узнать, что ничего нового нет. Умножьте это на сотни или тысячи посетителей, обращающихся к web-странице одновременно, и это может вылиться в ненужную нагрузку на ваш web-сервер.

Одно из возможных решений этой проблемы − это использование отправляемых сервером событий (server-sent events), что позволяет удерживать открытым подключение к web-серверу. Web-сервер может отправлять странице сообщения в любое время, и при этом отсутствует необходимость постоянно отключаться и подключаться к серверу и исполнять один и тот же сценарий. (Но если в этом есть необходимость, то можно использовать и опрос, т.к. отправляемые сервером события поддерживают эту возможность.)

Лучшее этой технологии в том, с какой легкостью используется система отправляемых сервером событий, а также в том, что она работает на большинстве web-хостов и исключительно надежная. Но это сравнительно новая система с соответствующим уровнем поддержки в браузерах.

source = new EventSource("server\_events.php");

source.onmessage = receiveMessage;

### Web-сокеты

**WebSocket** − протокол полнодуплексной связи (может передавать и принимать одновременно) поверх TCP-соединения, предназначенный для обмена сообщениями между браузером и web-сервером в режиме реального времени.

В настоящее время в W3C осуществляется стандартизация API Web Sockets. Черновой вариант стандарта этого протокола утверждён IETF.

WebSocket разработан для воплощения в web-браузерах и web-серверах, но он может быть использован для любого клиентского или серверного приложения. Протокол WebSocket − это независимый протокол, основанный на протоколе TCP. Он делает возможным более тесное взаимодействие между браузером и web-сайтом, способствуя распространению интерактивного содержимого и созданию приложений реального времени.

Если серверные события подразумевали постоянный запрос новой информации у сервера, то web-сокеты работают по принципу реального времени (двухсторонней связи):

socket.onopen = connectionOpen;

socket.onmessage = messageReceived;

socket.onerror = errorOccurred;

socket.onopen = connectionClosed;

Пример:

var socket = new WebSocket("ws://localhost/socketServer.php");

socket.send("UserName:alexerohinzzz@gmail.com");

socket.disconnect();

## **Другие API-интерфейсы**

### **Геолокация**

Браузер может вычислить ваше местонахождение, однако он не сообщит его web-сайту без вашего явного на то разрешения. Web-страница хочет получить данные о местоположении посетителя. Посетитель может разрешить предоставлять эти данные или же не предоставлять их. Такое поведение браузера является правилом стандарта геолокации, требующим, чтобы каждый web-сайт, пытающийся получить данные о местоположении посетителя, получил для этого его разрешение.

Чтобы вычислить местоположение посетителя, браузер заручается помощью поставщика услуг определения местоположения (location provider), например, для Chrome это сервис Google Location Services. Определение местоположения является задачей не из легких, и поставщик применяет несколько разных подходов для ее решения.

В случае настольного компьютера с фиксированным (т.е. небеспроводным) подключением, метод простой, но дает не очень точные результаты. Когда кто-то подключается к интернету, данные с его компьютера или локальной сети направляются (через кабель, выделенную линию или коммутируемое телефонное соединение) на мощное сетевое аппаратное устройство, которое, в свою очередь, направляет их в интернет. Это аппаратное устройство имеет однозначный IP-адрес, т.е. числовой код, который идентифицирует это устройство для других участников сети. Это устройство также имеет физический почтовый адрес.

Поставщик местоположения использует эти два типа информации для определения географического местоположения посетителя web-страницы. Сначала вычисляется IP-адрес устройства, через которое осуществляется подключение, а затем определяется его физический адрес. Естественно, такой непрямой подход позволяет определить точное местоположение не пользователя, а только его интернет-провайдера.

Метод определения местоположения посредством IP-адреса является самым неточным способом геолокации. Если имеется лучший источник данных, поставщик местоположения будет использовать этот источник.

Если посетитель зашел на страницу с нетбука или смартфона по беспроводному подключению, поставщик местоположения использует ближайшие точки беспроводного доступа. В идеальном случае поставщик местоположения выбирает данные из огромной базы данных, чтобы определить точное местонахождение этих точек доступа, а потом использует полученную информацию, чтобы определить местоположение посетителя методом триангуляции.

А в случае подключения с мобильного телефона поставщик местоположения применяет подобный метод триангуляции, но использует сигналы от разных антенн мобильной связи. Эта быстрая, сравнительно эффективная процедура обычно позволяет определить местоположение посетителя с точностью до 1 км.

Наконец, многие мобильные устройства оснащены специальными аппаратными средствами GPS (Global Positioning Service − глобальная система навигации и определения положения), что позволяет определять местоположение таких устройств с точностью всего лишь до нескольких метров. Но этот метод геолокации имеет свой недостаток — он медленнее и потребляет больше энергии, что важно для устройств, работающих на аккумуляторах. Кроме этого, он не особенно хорошо работает в городах с большими и высокими зданиями по причине отражения сигнала от строений.

Конечно же, возможны и другие способы определения местоположения. Ничто не препятствует поставщику местоположения использовать для этого другие источники информации, такие как данные от RFID-устройств (Radio Frequency Identification − радиочастотная идентификация), данные от устройств Bluetooth, файлы cookies с информацией от картографического сайта наподобие Google Maps и т.п.

Из всего этого можно сделать следующий вывод: независимо от способа подключения к интернету, даже если посетитель web-сайта сидит за настольным компьютером, возможность геолокации позволяет определить его местонахождение с большей или меньшей точностью. А если он выходит в интернет с мобильного телефона или с устройства, оснащенного аппаратным обеспечением GPS, его местоположение может быть определено с точностью скорее большей, чем меньшей:

navigator.geolocation.getCurrentPosition()

Возможность геолокации предельно проста. Она состоит из трех методов объекта navigator.geolocation: getCurrentPosition(), watchPosition() и clearWatch().

Объект navigator − это сравнительно незначительная часть JavaScript. Его несколько свойств предоставляют информацию о текущем браузере и его возможностях. Наиболее полезным из них является свойство navigator.userAgent, которое предоставляет информационную строку, содержащую подробные данные о браузере, его версии, а также операционной системы, в которой он выполняется.

Для получения местоположения посетителя вызывается метод getCurrentPosition(). Но следует понимать, что процесс определения местоположения занимает определенное время, в течение которого никакой уважающий себя браузер не остановит всю деятельность на странице, пока ожидает эти данные. Поэтому метод getCurrentPosition() исполняется асинхронно, т.е. после его запуска продолжается выполнение следующего за ним кода. Когда определение местоположения завершится, для обработки результатов активируется другой фрагмент кода.

Можно ожидать, что завершение определения местоположения сопровождается извещением, во многом подобном тому, как сообщается об окончании загрузки изображения или чтения текстового файла. Но язык JavaScript можно назвать каким угодно, но только не последовательным. И при вызове метода getCurrentPosition() ему передается функция завершения (completion function).

Далее приведен пример вызова этого метода:

navigator.geolocation.getCurrentPosition(

function(position) {

alert('Последний раз вас засекали здесь: ' +

position.coords.latitude + ", " + position.coords.longitude);

}

);

### **Фоновые вычисления**

В первое время после появления языка JavaScript никто особенно не беспокоился о его производительности. Этот язык был создан как простое средство для исполнения небольших сценариев в web-странице. Можно сказать, что он был необязательной примочкой, такая себе упрощенная версия языка Java для программистов-любителей.

20 лет спустя JavaScript завоевал интернет. Web-разработчики используют этот язык, чтобы добавить возможность интерактивности в web-страницы почти любого типа, от игр и картографических инструментов до интернет-магазинов и вычурных форм.

Один из примеров недостатков языка JavaScript − это его подход к выполнению большого объема вычислений. В большинстве современных систем программирования подобные задачи выполняются незаметно в фоновом режиме, позволяя пользователю продолжать работать над другими аспектами задачи. Но код JavaScript всегда исполняется на переднем плане. Поэтому любые трудоемкие вычисления прерывают выполнение всех других задач на странице, вынуждая пользователя ожидать их завершения. Игнорирование этой проблемы выльется вам в определенное количество очень недовольных посетителей, покинувших вашу страницу с твердым намерением больше на нее не возвращаться.

Изобретательные web-разработчики нашли несколько частичных решений данной проблемы. Эти решения основаны на разбиении долговременных задач на несколько меньших частей и исполнении этих частей по одной с помощью метода setInternval() или setTimeout().

Для некоторых типов задач это решение работает хорошо (например, это практичный способ для анимирования содержимого холста). Но если нужно выполнить одну очень долгую операцию безостановочно от начала до конца, этот метод порождает сложность и неразбериху.

Спецификация HTML5 предлагает лучшее решение в виде специализированного объекта, называющегося потоком (web worker), предназначенного для выполнения фоновых вычислений. Для выполнения долговременной задачи мы создаем новый поток, даем ему необходимый код и запускаем его выполнять поставленную задачу. В процессе выполнения потоком своей задачи с ним можно безопасно поддерживать ограниченное взаимодействие посредством текстовых сообщений.

var worker = new Worker("PrimeWorker.js");

worker.postMessage(myData);

Для того чтобы оценить достоинства потоков, нам нужно найти фрагмент кода подходящей трудоемкости. Использовать потоки для небольших задач нет смысла. Но для вычислений, напрягающих центральный процессор, которые могут подвесить браузер больше, чем на пару секунд, потоки существенно меняют дело. Возьмем, например, приложение, показанное на рисунке ниже, которое генерирует простые числа в определенном диапазоне. Хотя код для решения этой задачи простой, сама задача требует трудоемких вычислений, выполнение которых может занять значительное время

Установив диапазон, в котором нужно выполнять поиск простых чисел, можно запустить поиск, нажав кнопку "Начать поиск". Для сравнительно узкого диапазона (как на этом рисунке − от 1 до 200 000) задача выполняется в течение нескольких секунд, не причиняя особенных неудобств пользователю. Но установите более широкий диапазон (например, от 1 до 5 000 000), и страница перестанет реагировать в течение нескольких минут, если не больше. Пользователь не сможет щелкать мышью элементы страницы, прокручивать ее или выполнять какие-либо другие действия; кроме этого, браузер может выдать сообщение типа "долго исполняющийся сценарий" или залить серым цветом всю страницу.

Очевидно, что такую страницу можно улучшить с помощью потоков. Но прежде чем приступить к реализации этого улучшения, нам нужно вкратце ознакомиться с текущей разметкой и кодом JavaScript.

Эта разметка краткая и четкая. Страница использует два элемента управления <input>, для ввода начального и конечного числа диапазона. На ней имеется кнопка для запуска вычислений, а также два элемента <div> − один для вывода результатов, а другой для отображения сообщения о состоянии. Полностью вся разметка выглядит так:

<body>

<p>Задайте диапазон чисел от <input id="from" value="1"> до <input id="to" value="200000">.</p>

<button id="searchButton" onclick="doSearch()">Запустить поиск</button>

<div id="primeContainer">

</div>

<div id="status"></div>

</body>

### **История просмотров**

История просмотров − это функциональность HTML5, которая расширяет возможности объекта history JavaScript. Звучит просто, но нужно знать, когда и зачем следует использовать эту возможность.

Если вам никогда раньше не приходилось встречаться с объектом history, не стоит волноваться по этому поводу. До сих пор он не мог предложить нам ничего полезного. В действительности, традиционный объект history имеет только одно свойство и три основных метода. Это свойство length − содержит информацию о количестве элементов в списке истории просмотров (т.е. в поддерживаемом браузером списке недавно посещенных web-страниц). Вот пример использования этого свойства:

alert('У вас сохранено ' + history.length +

' страниц, в истории браузера.');

Наиболее полезным методом объекта history является метод back(). Этот метод позволяет переместить пользователя на один шаг назад в истории просмотров:

history.back();

Эффект этого метода равнозначен нажатию пользователем кнопки браузера "Назад". Подобным образом можно использовать метод forward() для перемещения на один шаг вперед или метод go() для перехода вперед или назад на определенное количество шагов.

Но все это не представляет большой ценности, если только вы не хотите создать для web-страницы собственные кнопки "Назад" и "Вперед". Но HTML5 добавляет этому объекту history несколько дополнительных возможностей, которые можно использовать для реализации намного более амбициозных задач.

Главной из этих возможностей является метод pushState(), который позволяет изменить URL в адресной строке браузера, не обновляя при этом содержимого страницы. Эта возможность приходится кстати в специфической ситуации, а именно при создании динамических страниц, которые незаметно загружают новое содержимое и плавно обновляют себя. В такой ситуации URL страницы и ее содержимое могут рассогласоваться.

Например, если страница загрузит в динамическом режиме содержимое с другой страницы, первоначальный URL страницы не изменится, что может вызвать разнообразные проблемы с созданием закладки для этой страницы. Эту проблему можно решить с помощью отслеживания истории сеансов

HTML5 предоставляет другое решение проблемы с URL в отслеживании истории сеансов. Можно изменять URL любым образом, не требуя при этом добавления в него странных символов, как в случае с методом hashbang. Например, когда страница exotic\_china.html загрузит пятый слайд, ее URL можно изменить так:

http://mysite.com/exotic\_china5.html

В этом случае браузер не будет пытаться запрашивать страницу exotic\_china5.html, а оставит прежнюю страницу, но загрузит для нее указанный слайд, а это нам и нужно. То же самое происходит, когда посетитель перемещается в обратном порядке в истории просмотра. Например, если посетитель перейдет к следующему слайду (и URL изменится на exotic\_china5.html), а потом возвратится назад к четвертому (возвращая URL к exotic\_china4.html), браузер сохраняет текущую страницу, и активирует событие, с помощью которого можно загрузить соответствующий слайд и восстановить правильную версию страницы.

Хотя с первого взгляда все выглядит как идеальное решение, в нем есть значительный недостаток. Чтобы эта система работала должным образом, для каждого используемого URL нужно в действительности создать страницу. Для нашего примера это означает, что надо создать страницы exotic\_china1.html, exotic\_china2.html, exotic\_china3.html и т.д. Ведь посетитель может захотеть обратиться к этим страницам напрямую, например, через закладку, введя адрес вручную, щелкнув по ссылке в сообщении электронной почты и т.п.

Для крупных web-сайтов (например, Facebook или Flickr) это не представляет большой проблемы, т.к. они могут использовать серверный код и предоставить содержимое одного и того же слайда в другой упаковке. Но самостоятельному web-разработчику для этого может потребоваться приложить несколько больше усилий.

Теперь, когда мы понимаем, каким образом история сеансов связана со страницами, собственно использование ее не представляет никаких трудностей. История сеансов состоит всего лишь из двух методов и одного события, добавленных к объекту history.

Наиболее важным является метод pushState(), который позволяет изменить часть URL, обозначающую страницу, в любое требуемое время. По причинам безопасности остальная часть URL изменениям не поддается. (Возможность изменять основную часть URL предоставила бы мощное средство для хакеров для подделывания web-сайтов.)

Метод pushState() принимает три аргумента, обязательным из которых является только третий − URL, выводящийся в строке адреса браузера. Первым параметром могут быть любые данные, сохраняемые для представления текущего состояния данной страницы. Как мы увидим далее, эти данные можно использовать, чтобы восстановить состояние страницы, если пользователь возвратится к данному URL посредством списка истории посещенных страниц браузера.

Второй параметр − это заголовок страницы, отображаемый в браузере. В настоящее время все браузеры дружно игнорируют эту подробность. Если нет надобности в установлении этих параметров, им можно просто присвоить значение null.

Далее приведен код, который нужно добавить в страницу exotic\_china.html, чтобы изменять ее URL в соответствии с текущим отображаемым слайдом:

function nextSlide() {

if (slideNumber == 5) {

slideNumber = 1;

} else {

slideNumber += 1;

}

// Добавленный код

history.pushState(slideNumber, null, "exotic\_china"

+ slideNumber + ".html");

//

goToNewSlide();

return false;

}

function previousSlide() {

if (slideNumber == 1) {

slideNumber = 5;

} else {

slideNumber -= 1;

}

// Добавленный код

history.pushState(slideNumber, null, "exotic\_china"

+ slideNumber + ".html");

//

goToNewSlide();

return false;

}

В качестве параметра состояния страницы используется номер текущего слайда. Важность этого обстоятельства станет понятной чуть позже, при рассмотрении события onPopState.

Используя метод pushState(), также следует иметь в виду событие onPopState, которое является его естественным дополнением. В то время как метод pushState() вставляет новый элемент в список История (History) браузера, событие onPopState предоставляет средство для обработки этого элемента, когда посетитель возвратится к нему.

Чтобы понять работу метода, рассмотрим, что происходит, когда посетитель просматривает все слайды. В процессе просмотра URL в адресной строке браузера меняется с exotic\_china.html на exotic\_china1.html, потом на exotic\_china2.html, на exotic\_china3.html и т.д. Хотя страница в действительности не изменяется, все эти URL добавляются в историю просмотра браузера.

Если пользователь щелкнет по ссылке для перехода на предыдущий слайд (например, с exotic\_china3.html на exotic\_china2.html), активируется событие onPopState. Это событие предоставляет коду информацию состояния, сохраненную ранее посредством метода pushState(). Задача программиста заключается в использовании этой информации, чтобы восстановить требуемую версию страницы. В настоящем примере это означает загрузку соответствующего слайда:

window.onpopstate = function(e) {

if (e.state != null) {

// Определяем номер слайда для данного состояния.

// (Этот номер также можно было вырезать из URL,

// используя свойство location.href, но для этого

// потребуется больше работы.)

slideNumber = e.state;

// Запрашиваем этот слайд у web-сервера

goToNewSlide();

}

}

Обратите внимание, что в этом примере выполняется проверка на наличие объекта состояния, прежде чем приступать к работе. Это делается из-за того, что некоторые браузеры (включая Chrome) активируют событие onPopState при начальной загрузке страницы, даже если метод pushState() еще не вызывался.

Существует еще один метод объекта history − replaceState(), но он используется не так часто. Метод replaceState() можно применять для того, чтобы заменить информацию о состоянии, которая связана с текущей страницей, не добавляя при этом ничего в список История (History).

## **Вопросы для самоконтроля и задачи**

1. Какие новые возможности появились в HTML5?
2. Как выглядит первая строка кода, написанного в HTML5?
3. Назовите новые API-интерфейсы, предложенные в спецификации HTML5.
4. Что такое глобальные (универсальные) атрибуты? Приведите несколько их примеров.
5. Для чего предназначены элементы <progress> и <meter>?
6. Для чего предназначен метод getCurrentPosition()?
7. Какие новые форматы HTML5 для дат и времени вы знаете?
8. Для чего предназначен метод pushState()?
9. Какие новые элементы для работы с web-формами появились в HTML5?
10. Какие атрибуты элемента <audio> вы знаете?

Глава 4. Адаптивная верстка



## **Основы адаптивной верстки**

Использование мобильных устройств растет во всем мире, с другой стороны, активно применяются 27- и 34-дюймовые мониторы. Растет и разрыв между наименьшими и наибольшими экранами, на которых просматривается web-содержимое. Благо есть решение, подходящее для постоянно расширяющейся арены браузеров и устройств. Нормальная работа сайта на множестве устройств и экранов возможна благодаря так называемому адаптивному дизайну, построенному на основе применения HTML5 и CSS3. Эта технология позволяет разметке и возможностям сайта подстраиваться под имеющееся окружение (размер экрана, тип ввода и возможности устройства или браузера).

Понятие «адаптивный web-дизайн» было введено Итаном Маркоттом в 2010 году. В статье на сайте www.alistapart.com/articles/responsive-web-design/ он свел воедино три уже существо­вавшие на тот момент технологии (гибкий макет на основе сетки, подстраиваемые по размеру изображения и элементы мультимедиа, а также медиазапросы) в еди­ный унифицированный подход, который он назвал адаптивным web-дизайном, то есть, адаптивный web-дизайн явля­ется представлением web-содержимого в наиболее удобном формате для окна про­смотра и устройства, обращающегося за этим содержимым.

Более того, адаптивный web-дизайн, создаваемый с помощью HTML5 и CSS3, может быть реализован без необходимости использования конечных решений на серверной стороне.

Зачастую люди путают два понятия − "резиновый" и "адаптивный" web-дизайн. Адаптивный web-дизайн подразумевает работу с различными макетами сайта в целях создания максимального удобства их использования на всевозможных типах устройств, учитывая всевозможные разрешения экранов.

Он «разумно» подгоняет общую картинку под экран в зависимости от размера дисплея на аппарате. Ну а резиновый дизайн − это, по сути, один из элементов адаптивного web-дизайна. Он, в свою очередь, лишь вносит изменения в пропорции web-страницы.

Таким образом, можно прийти к выводу о том, что адаптивный web-дизайн представляет собой комбинацию из резинового, а также прочих прогрессивных улучшений. Хотя из-за близких по духу определений порой и может возникать путаница.

Одним из наиболее популярных вариантов резинового дизайна web-страниц выступает двухколоночный макет, где в левой колонке располагаются ссылки, а в правой колонке − основной контент. Такая схема достаточно традиционна и привычна для пользователей, поэтому также пользуется признанием среди разработчиков (рис. 4.1).

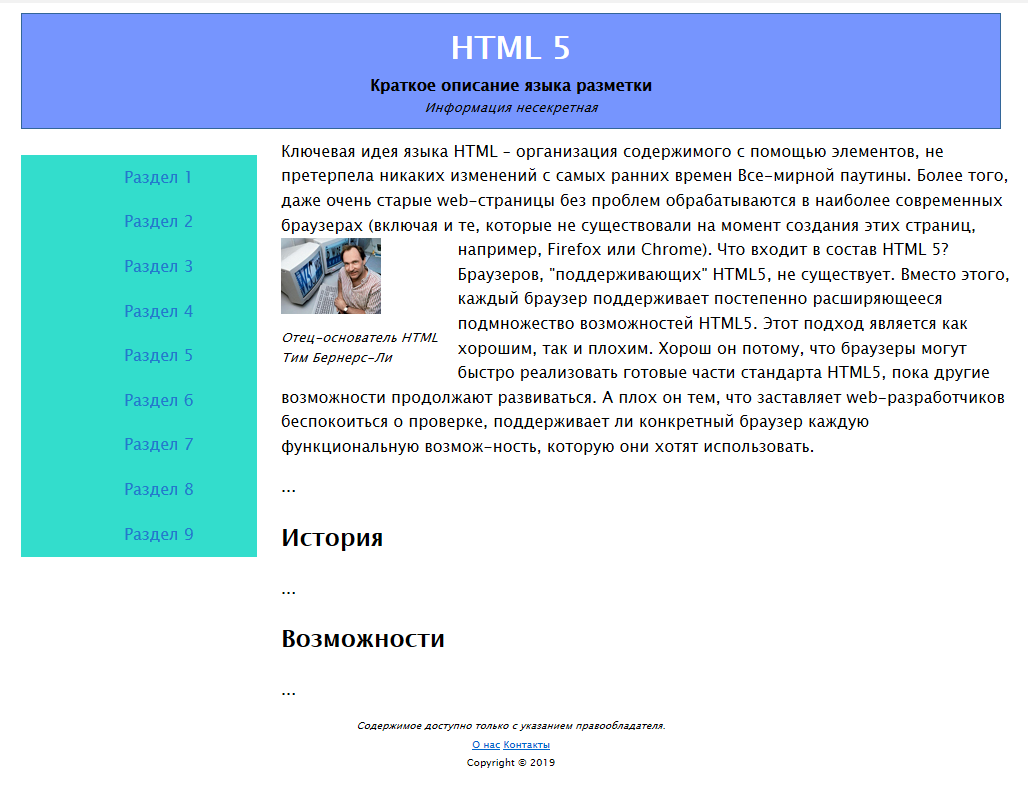


Рис. 4.1. Двухколоночный макет.

Создание двух колонок происходит с помощью стилевого свойства float со значением left, которое применяется для левой колонки. При этом одновременно следует установить отступ слева для правой колонки через margin-left, значение которого равно ширине левой колонки плюс расстояние между ними (пример 4.1).

**Пример 4.1.** Создание двух колонок

nav {

margin:10px;

max-width:1000px;

background-color:#3366CC;

text-align:center;

}

nav ul { /\* Левая колонка \*/

background-color:#33ddCC;

list-style:none;

float: left; /\* Обтекание справа \*/

width: 20%; /\* Ширина колонки \*/

}

nav ul a {

text-decoration:none;

display:block;

color:#2378cc;

padding: 10px 5px 10px 5px;

}

#rightcol {

/\* Правая колонка \*/

margin-left: 24%; /\* Отступ слева \*/

padding-left: 30px; /\* Поля слева \*/

}

Поскольку ширину правой колонки мы никак не ограничиваем, то она будет занимать все оставшееся пространство web-страницы.

## **Особенности адаптивного дизайна**

На ранней стадии развития наиболее характерным для адаптивного дизайна было построение, начинающееся с «рабочего стола», то есть дизайна с фиксированной шириной. Затем, чтобы этот дизайн работал на экранах меньшего размера, содержимое автоматически переформатировалось или удалялось. Но процесс не стоял на месте, и стало понятно, что все, от дизайна до содержимого и разработки, получается намного лучше, если действовать в обратном направлении, начиная с небольших экранов и работая по нарастающей.

Благодаря популярности и возможности повсеместного использования адаптивного web-дизайна теперь стало намного легче коммуницировать владельцу сайта с пользователями. Некоторые представления об адаптивном web-дизайне сложились уже у большинства людей. Понятие единого кода, способного работать практически на всех устройствах, становится весьма привлекательным.

При запуске проекта с адаптивным дизайном почти всегда возникает вопрос поддержки со стороны браузеров. При наличии столь широкого спектра браузеров и устройств вряд ли имеет смысл реализовывать полную поддержку изменений каждого отдельно взятого браузера. Как правило, чем старее браузер, тем больший объем работы и кода требуется для достижения желаемого результата или выравнивания эстетического восприятия с тем, которое пользователь получает при работе с современными браузерами. Поэтому рациональнее иметь менее вариативный и благодаря этому более быстродействующий код за счет распределения создаваемых впечатлений по уровням, обеспечивая получение самых совершенных визуальных эффектов и возможностей только на наиболее восприимчивых к ним браузерах.

За исключением экстремальных ситуаций, создаваемые сайты должны работать как минимум на каждом из самых распространенных браузеров. Кроме основных функциональных возможностей, для каждого web-проекта имеет смысл заранее определить, на каких платформах нужно в полной мере создать наилучшие впечатления от его работы, а на каких вполне возможно будет согласиться на визуальные или функциональные отступления.

На практике легче начинать с представления, создаваемого самым простым, базовым уровнем, и заниматься его усложнением (этот подход называется постепенным усложнением), чем подходить к решению проблемы с противоположной стороны − заниматься сначала созданием представления самого высокого уровня, а затем предпринимать попытки отступления для работы на платформах с более скромными возможностями (этот подход называется постепенным упрощением).

При рассмотрении вопроса о том, какие функции какими платформами и версиями браузеров поддерживаются, можно ознакомиться с сайтом http:// caniuse.com − там предоставляется весьма простой интерфейс для выявления, поддержкой какого браузера пользуются интересующие нас функции.

### **Инструментарий и текстовые редакторы для создания адаптивных web-конструкций**

Неважно, какими именно текстовым редактором или IDE-системой вы пользуетесь для создания своих адаптивных web-конструкций. Если самые простые текстовые редакторы позволяют вам успешно записывать код HTML, CSS и JavaScript, значит, все в порядке. Аналогично этому нет никакой особой оснастки, играющей важную роль в получении на выходе адаптивного web-дизайна. Фактически вам нужно лишь что-то позволяющее записывать код HTML, CSS и JavaScript. Чему вы отдадите предпочтение, Sublime Text, Vim, Coda, Visual Studio или Блокноту, не играет практически никакой роли. Работайте в той среде, которая вас больше всего устраивает.

Тем не менее следует знать, что, в отличие от прежних времен, сейчас имеется множество инструментальных средств (зачастую бесплатных), способных существенно облегчить выполнение рутинных, затратных по времени задач создания сайтов. Например, CSS-процессоры (Sass, LESS, Stylus, PostCSS) способны помочь с организацией кода, с переменными, в работе с цветовыми решениями и арифметикой. Такие средства, как PostCSS, способны также автоматизировать решение трудоемких и неприятных задач, например, установку в коде CSS префиксов производителей. Кроме того, проверочные средства могут сравнить ваш код HTML, JavaScript и CSS со стандартами, по которым вы работаете, сэкономив массу времени на выявление опечаток и синтаксических ошибок.

Постоянно появляются все новые и новые инструментальные средства с более совершенными свойствами. Поэтому, какие бы актуальные и полезные средства ни упоминались в настоящее время, следует иметь в виду, что где-то рядом на выходе уже есть что-то более интересное. Следовательно, в своих примерах мы не можем полагаться на что-либо иное, кроме стандартов на основе HTML и CSS. Начнем с простой структуры HTML5 (пример 4.2):

**Пример 4.2.** Пример HTML-документа без медиазапросов

<!doctype html>

<html class="no-js" lang="en">

<head>

<meta charset="utf-8">

<title>Our first responsive web page with HTML5 and CSS3</title>

<meta name="description" content="A basic responsive web page – an example from Chapter 1">

<link rel="stylesheet" href="CSS/styles.CSS">

</head>

<body>

<div class="Header">

<a href="/" class="LogoWrapper"><img src="img/SOC-Logo.png" alt="Scone O'Clock logo" /></a>

<p class="Strap">Scones: the most resplendent of snacks</p>

</div>

<div class="IntroWrapper">

<p class="IntroText">Occasionally maligned and

misunderstood; the scone is a quintessentially British classic.</p>

<div class="MoneyShot">

<img class="MoneyShotImg" src="img/scones.jpg" alt="Incredible scones" />

<p class="ImageCaption">Incredible scones, picture from Wikipedia</p>

</div>

</div>

<p>Recipe and serving suggestions follow.</p>

<div class="IntroInform">

<h3 class="SubHeader">IntroInform</h3>

<ul>

</ul>

</div>

<div class="HowToMake">

<h3 class="SubHeader">Method</h3>

<ol class="MethodWrapper">

</ol>

</div>

</body>

</html>

Изначально web-страницы обладают гибкостью. Если открыть страницу примера даже в теперешнем ее состоянии (без медиазапросов) и изменить размер окна браузера, текст будет подвергнут необходимой перекомпоновке. Эта проблема решается путем добавления в <head>контейнер следующего фрагмента кода:

<meta name="viewport" content="width=device-width">

Фактически этот метатег с именем viewport не считается стандартным способом указания браузеру способа отображения страницы (хотя и является стандартом де-факто). В данном случае наш метатег viewport представляет собой четкое предписание «отобразить содержимое во всю ширину экрана устройства».

Перед нами стоит несколько задача отображения изображения так, чтобы оно масштабировалось под различные размеры экрана, для этого в файл надо поместить код:

img {max-width: 100%;}

Правило, на котором основано свойство max-width, предполагает, что максимальная ширина всех изображений должна составлять 100 % их ширины (то есть они должны расширяться не более чем до 100 % своего размера). Когда содержащий изображения элемент (такой как body или div, внутри которого они находятся) меньше действительной ширины изображения, масштаб изображений будет просто подстроен, чтобы максимально занять доступное пространство.

А почему бы просто не воспользоваться свойством width: 100%? Чтобы превратить изображения в подстраиваемые, можно также применить более широко востребованное свойство width, например, width: 100%. Но в результате будет получен совершенно другой эффект. При использовании свойства width изображение будет показано с заданной шириной независимо от собственной ширины. В результате выполнения нашего примера получился бы логотип (также являющийся изображением), растянутый так, чтобы заполнить все 100 % своего контейнера. Когда контейнер намного шире изображения (как в случае с нашим логотипом), получается слишком растянутая картинка.

Теперь все располагается в соответствии с ожиданиями. Независимо от размера окна просмотра ничто теперь не выходит за границы страницы по горизонтали.

Но если посмотреть на страницу в более крупных окнах просмотра, основные стили как в прямом, так и в переносном смысле начинают восприниматься растянутыми. Фактически страница начинает выглядеть растянутой уже при ширине примерно 600 пикселов. На данном этапе было бы полезно получить возможность кое-что подправить. Может быть, изменить размер изображения и расположить его рядом с одной из сторон. Может быть, изменить размеры некоторых шрифтов и фоновые цвета элементов. Все эти функциональные возможности можно получить с помощью медиазапросов CSS.

Когда окно просмотра выходит по ширине за 600 пикселов, текущая разметка начинает казаться растянутой. Воспользуемся медиазапросами CSS3 для коррекции разметки в зависимости от ширины экрана. Медиазапросы позволяют применять конкретные CSS-правила на основе целого ряда условий (например, ширины и высоты экрана).

Понятие «контрольная точка» используется для определения точки, в которой адаптивный дизайн должен претерпеть существенные изменения. Когда только начинали применять медиазапросы, зачастую считалось, что контрольные точки в дизайне должны выстраиваться именно вокруг параметров наиболее популярных из имеющихся на то время устройств. Тогда эти контрольные точки обычно выстраивались вокруг параметров iPhone (320 × 480 пикселов) и iPad (768 × 1024 пикселов).

Но их выбор оказался неудачным, а в настоящее время он рассматривается как один из наихудших. Дело в том, что, поступая таким образом, мы ориентируем дизайн на конкретный размер экрана. А нам нужен адаптивный дизайн, то есть не то представление, которое неплохо смотрится только при конкретных размерах экрана, а то, которое не привязано к размеру экрана.

Поэтому позволим определять подходящие места для контрольных точек самому содержимому и дизайну. Может быть, исходная разметка начнет терять подобающий вид при ширине 500 пикселов и более, а может быть, 800 пикселов. Где именно нужно расставить контрольные точки, должно определяться дизайном вашего проекта.

Но чтобы наш простой пример не разрастался, мы сконцентрируемся на одном типе медиазапроса, касающегося минимальной ширины. Правила CSS внутри этого типа медиазапроса применяются только в том случае, если окно просмотра имеет минимальную заданную ширину. Точная минимальная ширина может указываться с применением целого набора различных единиц измерения длины, включая проценты, em, rem и px (пиксел). В CSS медиазапрос минимальной ширины записывается следующим образом:

@media screen and (min-width: 50em) {

/\* стили \*/

}

Директива @media сообщает браузеру о начале медиазапроса, компонент screen (применять это объявление экрана в данной ситуации технически не обязательно) сообщает браузеру, что правила должны применяться ко всем типам экранов, и компонент and (min-width: 50em) сообщает браузеру, что правила должны действовать для всех окон просмотра, чья ширина превышает 50em.

Для того, чтобы скорректировать код под экран большего размера, определим разную разметку для разных размеров окна просмотра. Остановим чрезмерное разрастание соответствующего теме сайта изображения, удерживая его в правой части экрана. Затем можно поместить вводный текст в левую часть экрана. Все эти изменения могут быть получены относительно легко, нужно лишь поместить конкретные стили в медиазапрос. На экранах меньшего размера страница будет выглядеть так же, как и раньше, но как только окно просмотра окажется больше или равно 50rem, страница будет настраиваться под новую разметку. Добавленные стили разметки имеют следующий вид:

@media screen and (min-width: 50em) {

.IntroWrapper

{display: table;

table-layout: fixed;

width: 100%;

}

.MoneyShot,.IntroText

{display: table-cell;

width: 50%;

vertical-align: middle;

text-align: center;

}

.IntroText

{padding: .5em;

font-size: 2.5rem;

text-align: left;

}

.IntroInform

{font-size: .9em;

float: right;

padding: 1em;

margin: 0 0 .5em 1em;

border-radius: 3px;

background-color: #ffffdf;

border: 2px solid #e8cfa9;

}

Таким образом, добавлением всего лишь нескольких дополнительных стилей, создана страница, реагирующая на размер окна просмотра и предлагающая по мере необходимости желательную разметку.

## **Медиазапросы**

Медиазапросы CSS3 позволяют нацеливать конкретные CSS-стили на определенные возможности устройств или возникающие ситуации. Если углубиться в W3C спецификацию, относящуюся к модулю медиазапросов CSS3 (http://www.w3.org/ TR/CSS3-mediaqueries/), можно увидеть их следующее официальное представление: «Медиазапрос состоит из типа среды и выражений в количестве от нуля и более, которые ведут проверку условий конкретных медиасвойств. К медиасвойствам, используемым в медиазапросах, относятся ширина − width, высота − height и цвет − color. За счет использования медиазапросов представления без изменения своего содержимого могут быть привязаны к конкретному диапазону устройств вывода информации».

Без медиазапросов мы не сможем внести существенные изменения в визуальное представление сайта с помощью одних только CSS-таблиц. Они облегчают нам написание стабильных CSS-правил, упреждающих такие случаи, как портретная ориентация экрана, небольшие или большие размеры окна просмотра и многое другое. Хотя подстраиваемая разметка может справляться с выдерживанием определенного дизайна в довольно широком диапазоне изменений с учетом всего диапазона размеров экрана, который мы надеемся охватить, бывают случаи, когда требуется более глубокий пересмотр разметки. Возможности такого пересмотра дают медиазапросы. Их нужно рассматривать в качестве основной условной логики для CSS. Основная условная логика в CSS. Во всех настоящих языках программирования имеются средства, благодаря применению которых обслуживается одна или несколько возможных ситуаций. Обычно такие средства существуют в виде условной логики, примером которой может послужить инструкция if – else.

Медиазапросы являются одним из механизмов CSS, позволяющих нам создавать основную условную логику. В том случае, когда при использовании медиазапроса складываются конкретно оговоренные в нем условия, в область видимости попадают именно те стили, которые в нем объявляются.

**Синтаксис медиазапроса**

Так как же выглядят медиазапросы и, что более важно, как они работают? Рассмотрим пример:

body { background-color: grey; }

@media screen and (min-width: 320px) {

body { background-color: green; }

}

@media screen and (min-width: 550px) {

body { background-color: yellow; }

}

@media screen and (min-width: 768px) {

body { background-color: orange; }

}

@media screen and (min-width: 960px) {

body { background-color: red; }

}

Теперь посмотрите на файл в браузере и измените размер окна. Фоновый цвет страницы будет меняться в зависимости от текущего размера окна просмотра.

**Медиазапросы в тегах link**

Все, кому приходилось работать с CSS, начиная со второй, знают о возможности указывать тип устройства (например, screen или print) применительно к таблице стилей с атрибутом media тега <link>. Рассмотрим следующий пример, который следует помещать в теги <head> своей разметки:

<link rel="style sheet" type="text/CSS" media="screen" href="screenstyles.CSS">

Медиазапросы добавляют к возможности простого указания типа устройства возможность указания целевого назначения стилей на основе возможностей или характерных особенностей устройства. Их нужно рассматривать в качестве вопросов к браузеру. Если браузер даст положительный ответ, применяются стили, заключенные в медиазапрос. Если ответ будет отрицательным, эти стили применяться не будут. Вместо того чтобы просто спросить: «Это экран?» − все, что по максимуму можно сделать, используя только CSS2, − медиазапросы задают более сложные вопросы. С их помощью можно спросить: «Экран ли это и просматривается ли он в портретной ориентации?» Рассмотрим соответствующий пример:

<link rel="stylesheet" media="screen and (orientation: portrait)" href="portrait-screen.CSS" />

Сначала с помощью выражения медиазапроса задается вопрос о типе (экран ли это?), а затем о свойстве (находится ли экран в портретной ориентации?). Таблица стилей portrait-screen. CSS будет применяться для любого экранного устройства с портретной ориентацией и игнорироваться для любых других вариантов устройств и свойств. Логику любого выражения медиазапроса можно поменять на противоположную, добавляя вначале отрицание not. Например, результат нашего предыдущего примера будет прямо противоположным, и файл с таблицей стилей будет применяться для всего, что не является экраном с портретной ориентацией:

<link rel="stylesheet" media="not screen and (orientation: portrait)" href="portrait-screen.CSS" />

**Объединение медиазапросов**

В одной строке можно также выстроить сразу несколько выражений. Давайте, к примеру, расширим один из предыдущих фрагментов кода и вдобавок ко всему прочему ограничим применение файла устройствами, ширина окна просмотра которых составляет не менее 800 пикселов:

<link rel="stylesheet" media="screen and (orientation: portrait) and (min-width: 800px)" href="800wide-portrait-screen.CSS" />

К тому же у нас может быть список медиазапросов. Файл будет применен, если будет получен положительный ответ на любой из перечисленных запросов. Если положительных ответов не окажется, файл применен не будет. Рассмотрим следующий пример:

<link rel="stylesheet" media="screen and (orientation: portrait) and (min-width: 800px), projection" href="800wide-portrait-screen.CSS" />

В нем есть две интересные особенности. Во-первых, для разделения медиазапросов используются запятые. Во-вторых, следует заметить, что в скобках после ключевого слова projection отсутствует следующая пара «свойство — значение» и/или сочетание свойств и значений. Дело в том, что при отсутствии таких значений медиазапрос применяется ко всем устройствам, относящимся к данному типу медиаустройств. В нашем примере стили будут применяться ко всем проекторам.

Следует иметь в виду, что для указания медиазапросов можно использовать любую единицу измерения длины, применяемую в CSS. Наиболее часто используются пикселы (px), но подойдут также em и rem. Таким образом, если нужна контрольная точка на 800 пикселах, но указанная в em, нужно просто разделить количество пикселов на 16. Например, 800 пикселов можно также указать как 50 em (800 / 16 = 50).

**Медиазапросы с использованием @import**

Для условной загрузки таблиц стилей в существующую таблицу стилей можно использовать также CSS-конструкцию @import. Например, следующий код импортирует таблицу стилей под названием phone.CSS при условии, что устройством является экран с максимальной шириной окна просмотра 360 пикселов:

@import URL("phone.CSS") screen and (max-width:360px);

Следует помнить, что при использовании CSS-конструкции @import код добавляется к HTTP-запросам, что влияет на скорость загрузки, поэтому данный метод нужно использовать осмотрительно.

**Медиазапросы в CSS**

До сих пор медиазапросы включались нами в виде ссылок на CSS-файлы, а сами ссылки помещались в раздел <head></head> кода HTML и в виде инструкций @import. Но, скорее всего, нам захочется использовать медиазапросы внутри самих таблиц стилей. Например, если в таблицу стилей добавить следующий код, все элементы h1 приобретут зеленый цвет при условии, что ширина экрана устройства − не более 400 пикселов:

@media screen and (max-device-width: 400px) {

h1 { color: green }

}

Сначала в виде правила @media указывается, что нам нужен медиазапрос, затем задается желаемый тип, с которым требуется совпадение. В предыдущем фрагменте кода требуется применить правила, относящиеся только к экрану, но, к примеру, не к принтеру. Затем внутри скобок вводятся особенности запроса. После этого, как и в любом CSS-правиле, открываются фигурные скобки, внутри которых записываются требуемые стили. Здесь, видимо, следует заметить, что в большинстве ситуаций указывать экран (screen) не нужно. Основное положение спецификации звучит следующим образом: «Сокращенный синтаксис предлагается для тех медиазапросов, которые применяются ко всем типам медиаустройств; ключевое слово all может быть опущено (наряду со следующим за ним and). То есть если тип медиаустройства не задан конкретным образом, подразумевается настройка all».

Следовательно, пока не возникнет потребность нацелить стили на конкретные типы медиаустройств, screen и and можно опустить. Впредь все медиазапросы в примерах будут записываться с учетом этой возможности.

**Что можно тестировать с помощью медиазапросов**

При построении адаптивного дизайна чаще всего используются медиазапросы, относящиеся к ширине окна просмотра (width), перечень же всех возможностей Media Queries Level 3, которыми можно будет воспользоваться, представлен ниже:

width: − ширина окна просмотра;

height: − высота окна просмотра;

device-width: − ширина поверхности отображения (для нас это, как правило, ширина экрана устройства);

device-height: − высота поверхности отображения (для нас это, как правило, высота экрана устройства);

orientation: − возможность проверки портретной или альбомной ориентации устройства;

aspect-ratio: − соотношение ширины к высоте на основе ширины и высоты окна просмотра. Дисплей с соотношением сторон 16:9 может быть описан как aspectratio: 16/9;

device-aspect-ratio: − эта возможность аналогична предыдущей, но основывается на ширине и высоте не окна просмотра, а поверхности отображения устройства;

color: − количество битов, приходящееся на каждую составляющую цвета. Например, min-color: 16 задаст проверку того, обладает ли устройство цветом с глубиной 16 бит;

color-index: − количество записей в таблице поиска цветов устройства. Значения должны быть числовыми и не могут быть отрицательными;

monochrome: − возможность проверки количества битов на пиксел в буфере монохромного кадра. Значение должно быть целым числом, например monochrome: 2, и не может быть отрицательным;

resolution: − эта возможность может использоваться для проверки разрешения экрана или принтера, например min-resolution: 300dpi. Может также приниматься единица измерения в точках на сантиметр, например min-resolution: 118dpcm;

scan: − это свойство может отображать значение развертки (прогрессивной или чересстрочной), которое имеет отношение в основном к телевизионным устройствам. Например, нацеливание на устройство с параметрами 720p HD TV (буква «p» в 720p означает progressive − «прогрессивная») может быть обозначено с помощью выражения scan: progressive, a на устройство 1080i HD TV (буква «i» в 1080i означает interlaced − «чересстрочная») − с помощью выражения scan: interlace;

grid: − эта возможность показывает, на какой основе построено устройство, сеточной или растровой. Все перечисленные возможности, за исключением scan и grid, для создания диапазонов могут использоваться с префиксом min или max. Рассмотрим, к примеру, следующий фрагмент кода:

@import URL("tiny.CSS") screen and (min-width:200px) and (max-width:360px);

Здесь минимум (min) или максимум (max) применены для задания диапазона ширины − width. Файл tiny.css может быть импортирован для экранных устройств с минимальной шириной окна просмотра 200 пикселов и максимальной шириной окна просмотра 360 пикселов.

**Медиасвойство использования сценариев**

Согласно устоявшейся практике тегу HTML присваивается класс, показывающий исходное отсутствие кода JavaScript с последующей заменой этого класса на другой при запуске JavaScript. Тем самым предоставляется легко реализуемая возможность разветвления кода, включая CSS, на основе этого нового класса, присваиваемого тегу HTML. В частности, используя этот прием, можно написать правила, применяемые к конкретным пользователям, у которых включен JavaScript. Чтобы не запутаться, рассмотрим пример кода. В самом начале это будет тег, созданный в HTML:

<html class="no-js">

Когда на странице запустится JavaScript, одной из первых задач станет замена класса no-js:

<html class="js">

После того как это будет сделано, можно написать конкретные CSS-правила, применимые исключительно при наличии JavaScript, например .js .header { display: block; }.

А вот медиасвойство использования сценариев, имеющееся в спецификации CSS Media Queries Level 4, предназначено для обеспечения более соответствующего общему стандарту способа достижения аналогичного результата непосредственно в коде CSS:

@media (scripting: none) { /\* стили, предназначенные для применения в отсутствие JavaScript \*/ }

И при наличии JavaScript:

@media (scripting: enabled) { /\* стили, предназначенные для применения при наличии JavaScript \*/ }

И наконец, это свойство предназначено также для выявления факта применения JavaScript только при начальном отображении страницы. В спецификации W3C приводится пример задания отображаемой страницы, которая может быть размечена при начальном выводе на экран, после чего JavaScript на ней использоваться не будет:

@media (scripting: initial-only) { /\* стили для страницы, применяющей JavaScript только при ее начальном выводе \*/ }

**Медиасвойства, связанные с взаимодействием со страницей**

В W3C медиасвойство указателя pointer, представлено следующим образом: «Медиасвойство pointer используется для запроса наличия и точности указывающего устройства, например мыши. Если у устройства имеется несколько механизмов ввода, медиасвойство pointer должно отражать характеристики первичного механизма ввода в соответствии с определением пользовательского агента (user agent)».

Существует три возможных состояния указателя pointer: none (отсутствует), coarse (грубый) и fine (тонкий).

В качестве устройства указателя с параметром coarse может оказаться палец на устройстве с сенсорным экраном. Но это может быть и курсор игровой консоли, не обладающий такой же высокой точностью, как указатель мыши:

@media (pointer: coarse) {

/\* стили на случай присутствия указателя, имеющего состояние coarse \*/

}

Устройством со свойством pointer, имеющим значение fine, может быть мышь, а также стилус-перо или любой другой указательный механизм высокой точности:

@media (pointer: fine) {

/\* стили на случай присутствия указателя с высокой точностью \*/

}

Разумнее всегда предполагать, что пользователи работают с устройством сенсорного ввода, и задавать размеры элементов пользовательского интерфейса соответствующим образом. Тогда, даже если они используют мышь, с интерфейсом у них не возникнет никаких затруднений. Если же предположить, что устройством ввода является мышь, возникнут осложнения из-за невозможности надежного определения касания при работе с элементами интерфейса.

**Медиасвойство hover**

Тестирует пользовательскую возможность проведения указателя над элементами экрана. Если пользователь располагает несколькими устройствами ввода (например, сенсором и мышью), используются характеристики основного устройства ввода. Далее перечислены возможные значения и показаны примеры кода. Для пользователей, не имеющих возможности провести указатель над элементами экрана, можно нацелить стили для значения этого свойства, равного none:

@media (hover: none) {

/\* стили для тех случаев, когда провести указатель над элементами невозможно \*/

}

Для тех пользователей, которые имеют такую возможность, но должны для ее инициирования предпринять определенные действия, можно воспользоваться значением on-demand:

@media (hover: on-demand) {

/\* стили для тех случаев, когда пользователь имеет возможность провести указатель над элементами, но от него требуются определенные действия \*/

}

Для пользователей, имеющих возможность провести указатель над элементами, может быть использовано объявление hover без указания какого-либо значения:

@media (hover) {

/\* стили для тех случаев, когда пользователь имеет возможность провести указатель над элементами \*/

}

Существуют также медиасвойства any-pointer и any-hover. Они похожи на рассмотренные ранее hover и pointer, но проверяют возможности любого из возможных устройств ввода.

**Медиасвойства среды**

Медиасвойства среды предназначены для изменения дизайна на основе свойств окружающей среды, например, уменьшить уровень освещенности при переходе пользователя в хуже освещенное помещение или повысить контрастность и использовать более яркое цветовое оформление при переходе в место с ярким солнечным светом, например:

@media (light-level: normal) {

/\* стили для стандартной освещенности \*/

}

@media (light-level: dim) {

/\* стили для приглушенной освещенности \*/

}

@media (light-level: washed) {

/\* стили для яркой освещенности \*/

}

## **Библиотека Bootstrap**

Создание адаптивного дизайна с помощью медиазапросов является довольно трудоемкой задачей, поскольку требует от разработчика по сути сделать несколько полных наборов css-правил для различных размеров экранов. Объем работы возрастает пропорционально количеству поддерживаемых устройств.

В связи с этим появились готовые библиотеки, позволяющие с некоторыми ограничениями довольно быстро создавать адаптивные макеты. Одной из таких библиотек является библиотека Bootstrap, в основе которой лежит идея о том, что типовой сайт может состоять из 12 колонок, которые можно гибко перемещать в зависимости от размеров экрана.

Bootstrap − это свободный набор инструментов для создания [сайтов](https://ru.wikipedia.org/wiki/%D0%A1%D0%B0%D0%B9%D1%82) и [web-приложений](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5). Bootstrap включает в себя [HTML](https://ru.wikipedia.org/wiki/HTML)- и [CSS](https://ru.wikipedia.org/wiki/CSS)-шаблоны оформления для [типографики](https://ru.wikipedia.org/wiki/%D0%A2%D0%B8%D0%BF%D0%BE%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D0%BA%D0%B0), web-форм, кнопок, меток, блоков навигации и прочих компонентов web-интерфейса, включая [JavaScript](https://ru.wikipedia.org/wiki/JavaScript)-расширения. Bootstrap использует современные наработки в области CSS и HTML.

### Подключение библиотеки

Чтобы подключить данную библиотеку нужно зайти на [getbootstrap.com](http://getbootstrap.com/), скачать zip-архив целиком либо сформировать список элементов и компонентов, которые необходимо применять в работе. Также можно настроить цвета, формы и шрифты. После того как выбор сделан, нажмите «Compile and Download» для загрузки архива с файлом bootstrap.min.css и уже минифицированным bootstrap.min.css, скомпилированным на основе ваших настроек. Такая оптимизация на уровне выбора элементов, да еще и с минификацией, позволяет значительно уменьшить объем получаемых файлов.

Получаемый файл подключается следующим образом:

<link rel='stylesheet' href='/css/bootstrap.min.css' type='text/css' media='all'>

Если вам нужен будет jQuery на сайте, то не забудьте подключить и его:

<script src="http://code.jquery.com/jquery-latest.min.js"></script>

Либо подключите последнюю версию с Google API или скачайте библиотеку себе на сайт и подгружайте со своего хостинга. Затем можно добавить и .js файл самого Bootstrap:

<script src="js/bootstrap.min.js"></script>

Данный скрипт нужен для того, чтобы сохранить совместимость на старых версиях IE:

<script src="js/respond.min.js"></script>

Теперь Bootstrap подключен и настроен.

**Основные преимущества Bootstrap:**

1. Экономия времени − потому что используются уже готовые классы и дизайн.
2. Адаптивность, высокая скорость и оптимизация, стандартизация интерфейсов − динамичные макеты Bootstrap качественно отображаются на самых разных устройствах без необходимости внесения изменений в разметку.
3. Дизайн − единые шаблоны и стилевое оформление элементов макета и всех страниц на сайте в целом. И при этом Bootstrap кросс-браузерный и хорошо отображается во всех браузерах Safari, Firefox, IE, EDGE и написанных на основе Chromium.
4. Простота и открытость − можно использовать Bootstrap просто, с ним справляются начинающие web-разработчики.

Также в Bootstrap используется динамический язык стилей LESS, которые расширяет возможности CSS − разработчики могут управлять цветами, создавать вложенные колонки и переменные.

Развитие интернета и web-сайтов показало, что в 2016 году половина пользователей интернета посещает сайты с мобильных устройств и небольших экранов. Это значит, что современный web-разработчик должен думать о функционировании сайта не только на стационарных устройствах, но и на смартфонах и планшетах с тачскрином.

Создание отдельной мобильной версии для сайта конечно выход, но тогда потребуется делать в два раза больше работы на разработку и поддержание кода, а это не всегда экономически эффективно. Это может позволить себе крупная компания, которая для мобильных пользователей делает еще и приложение для смартфонов.

Концепция отзывчивого web-дизайна, воплощенного в фреймворке Bootstrap, решает именно эту проблему: сайт одинаково «отзывается» и отображает информацию наиболее полным образом вне зависимости от типа экрана и размера устройства. Содержание и цветовая гамма не меняются, меняется только форма и способ сгруппировать информационные и навигационные блоки сайта наиболее удобным для пользователя способом.

### Сетка

Bootstrap включает в себя гибкую мобильную систему с первой динамичной сеткой, которая соответствующим образом масштабируется до 12 столбцов по мере увеличения размера устройства или области просмотра. Также он включает в себя [предопределенные классы](https://getbootstrap.com/docs/3.3/css/#grid-example-basic) для удобных опций макета, а также мощные [средства для создания более семантических макетов](https://getbootstrap.com/docs/3.3/css/#grid-less).

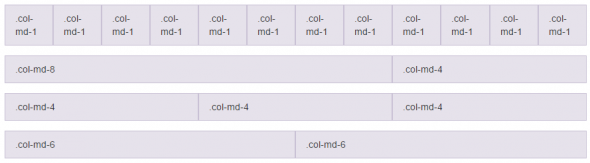


Рис. 4.5. 12-колоночная сетка Bootstrap.

Сетка используется для создания макетов страниц через ряд строк и столбцов, в которых размещается ваш контент. Вот как работает сеточная система Bootstrap:

* Строки должны быть размещены в пределах класса .container (фиксированной ширины) или класса .container-fluid (полной ширины) для правильного выравнивания и заполнения.
* Для создания горизонтальных групп столбцов используются строки.
* Содержимое должно быть помещено в столбцы, и только столбцы могут быть непосредственными потомками строк.
* Предопределенные классы сетки, такие как .row и .col-xs-4 доступны для быстрого создания макетов сетки.
* Для задания расстояния между столбцами можно исопльзовать padding.
* Столбцы сетки создаются путем указания числа двенадцати доступных столбцов, которые вы хотите охватить. Например, три равных столбца будут использовать три .col-xs-4.
* Если в одной строке размещено более 12 столбцов, каждая группа дополнительных столбцов будет перенесена на новую строку.
* Классы сетки применяются к устройствам с шириной экрана, превышающей или равной размерам точек останова, и переопределяют классы сетки, предназначенные для небольших устройств. Следовательно, например, применение какого-либо .col-md-\*класса к элементу повлияет не только на его стиль на средних устройствах, но и на больших устройствах, если .col-lg-\*класс отсутствует.

**Медиа-запросы**

Для создания ключевых точек в сеточной системе используются следующие медиа-запросы в файлах Less:

/\* Extra small devices (phones, less than 768px) \*/

/\* No media query since this is the default in Bootstrap \*/

/\* Small devices (tablets, 768px and up) \*/

@media (min-width: @screen-sm-min) { ... }

/\* Medium devices (desktops, 992px and up) \*/

@media (min-width: @screen-md-min) { ... }

/\* Large devices (large desktops, 1200px and up) \*/

@media (min-width: @screen-lg-min) { ... }

**Варианты сетки**

Как можно использовать возможности системы сетки Bootstrap (таблица 4.1).

**Таблица 4.1.** Сетка Bootstrap

|  | Очень маленькие устройства  (телефоны) (<768 пикселей) | Небольшие устройства  (планшеты)  (≥768px) | Средние устройства  (ноутбуки) (≥992px) | Большие  устройства  (настольные ПК) (≥1200px) |
| --- | --- | --- | --- | --- |
| Поведение сетки | Горизонтально | Свернуто, будет горизонтальным по достижении контрольных точек | | |
| Ширина контейнера | Нет (авто) | 750px | 970px | 1170px |
| Префикс класса | .col-xs- | .col-sm- | .col-md- | .col-lg- |
| Количество столбцов | 12 | | | |
| Ширина колонки | Авто | ~ 62px | ~ 81px | ~ 97px |
| Ширина отступа | 30px (15px на каждой стороне столбца) | | | |
| Может быть вложенным | да | | | |
| Упорядочивание колонок | да | | | |

Используйте классы колонок со специальными контрольными точками (например, .col-sm-6) для легкого расположения колонок без использования явно обозначенных номеров классов.

Например, рассмотрим две сетки, которые подойдут к любому устройству и зоне видимости, от xs до xl:

<div class="container">

<div class="row">

<div class="col">

1 из 2

</div>

<div class="col">

2 из 2

</div>

</div>

<div class="row">

<div class="col">

1 из 3

</div>

<div class="col">

2 из 3

</div>

<div class="col">

3 из 3

</div>

</div>

</div>

Авто-расположение колонок в сетке flexbox также дает возможность установить ширину одной колонки, при этом остальные «родственные» колонки автоматически изменят свой размер вокруг нее. Заметим, что другие колонки будут изменять размер независимо от ширины центральной колонки:

<div class="container">

<div class="row">

<div class="col">

1 из 3

</div>

<div class="col-6">

2 из 3 (широкая)

</div>

<div class="col">

3 из 3

</div>

</div>

<div class="row">

<div class="col">

1 из 3

</div>

<div class="col-5">

2 из 3 (широкая)

</div>

<div class="col">

3 из 3

</div>

</div>

</div>

Используйте классы col-{breakpoint}-auto для создания колонок, изменяющих свою ширину по ширине содержимого:

<div class="container">

<div class="row justify-content-md-center">

<div class="col col-lg-2">

1 из 3

</div>

<div class="col-md-auto">

Содержимое адаптивной ширины

</div>

<div class="col col-lg-2">

3 из 3

</div>

</div>

<div class="row">

<div class="col">

1 из 3

</div>

<div class="col-md-auto">

Содержимое адаптивной ширины

</div>

<div class="col col-lg-2">

3 из 3

</div>

</div>

</div>

Для сеток, которые выглядят и располагаются одинаково на всех устройствах любого размера, используйте классы .col и .col-\*. Определите именованный класс с цифрой, когда вам нужна колонка определенного размера, во всех остальных случаях свободно пользуйтесь .col:

<div class="container">

<div class="row">

<div class="col">col</div>

<div class="col">col</div>

<div class="col">col</div>

<div class="col">col</div>

</div>

<div class="row">

<div class="col-8">col-8</div>

<div class="col-4">col-4</div>

</div>

</div>

Используя один набор из классов .col-sm-\*, вы можете создать базовую сетку, которая вначале сложена по вертикали, а потом ее колонки становятся горизонтальными:

<div class="container">

<div class="row">

<div class="col-sm-8">col-sm-8</div>

<div class="col-sm-4">col-sm-4</div>

</div>

<div class="row">

<div class="col-sm">col-sm</div>

<div class="col-sm">col-sm</div>

<div class="col-sm">col-sm</div>

</div>

</div>

Если необходимо изменить порядок следования колонок, то используйте комбинацию разных классов для каждого яруса:

<div class="container">

<div class="row">

<div class="col-12 col-md-8">.col-12 .col-md-8</div>

<div class="col-6 col-md-4">.col-6 .col-md-4</div>

</div>

<div class="row">

<div class="col-6 col-md-4">.col-6 .col-md-4</div>

<div class="col-6 col-md-4">.col-6 .col-md-4</div>

<div class="col-6 col-md-4">.col-6 .col-md-4</div>

</div>

<div class="row">

<div class="col-6">.col-6</div>

<div class="col-6">.col-6</div>

</div>

</div>

### Компоненты

Bootstrap предоставляет значительное количество готовых компонентов таких как: графические значки, выпадающие меню, группы ввода, навигационные панели, сообщения для предупреждений и многие другие. Рассмотрим некоторые из них.

**Выпадающее меню**

Выпадающее меню – это переключаемое, контекстное меню для отображения списка ссылок. Его интерактивность обеспечивается JavaScript-плагином для выпадающего меню. Для его применения необходимо использовать класс .dropdown для выпадающего меню и списка. Также можно использовать другой элемент, который задает position: relative:

<div class="dropdown">

<button class="btn btn-default dropdown-toggle" type="button" id="dropdownMenu1" data-toggle="dropdown" aria-haspopup="true" aria-expanded="true">

Dropdown

<span class="caret"></span>

</button>

<ul class="dropdown-menu" aria-labelledby="dropdownMenu1">

<li><a href="#">Action</a></li>

<li><a href="#">Another action</a></li>

<li><a href="#">Something else here</a></li>

</ul>

</div>

В результате будет получен следующий результат (см. рисунок 4.6).



Рис. 4.6. Пример выпадающего меню.

**Группы иконок**

Bootstrap позволяет объединить кнопки в одну линию, создав таким образом группу кнопок. С помощью JavaScript и плагина кнопок, можно еще добавить стиль поведения переключателей или checkbox. Оберните серию кнопок, содержащих класс .btn, классом .btn-group:

<div class="btn-group" role="group" aria-label="...">

<button type="button" class="btn btn-default">Left</button>

<button type="button" class="btn btn-default">Middle</button>

<button type="button" class="btn btn-default">Right</button>

</div>

Результат представлен на рисунке 4.7.

C:\Users\1\AppData\Local\Microsoft\Windows\INetCache\Content.Word\5.jpg

Рис. 4.7. Пример группы иконок.

**Кнопка выпадающего меню**

Используйте любую кнопку для выпадающего меню, разместив ее в .btn-group и обеспечив ее меню:

<!-- Одна кнопка -->

<div class="btn-group">

<button type="button" class="btn btn-default dropdown-toggle" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">

Action <span class="caret"></span>

</button>

<ul class="dropdown-menu">

<li><a href="#">Action</a></li>

<li><a href="#">Another action</a></li>

<li><a href="#">Something else here</a></li>

<li role="separator" class="divider"></li>

<li><a href="#">Separated link</a></li>

</ul>

</div>

Будет получен следующий результат (см. рисунок 4.8).



Рис. 4.8. Пример выпадающего меню.

**Группы формы ввода**

Группирование форм ввода позволяет добавлять текстовые данные или кнопки до, после или с обеих сторон к любому текстовому элементу input. Используйте .input-group с .input-group-addon или .input-group-btn для добавления в начало или в конец элементов к текстовому полю с классом .form-control:

<div class="input-group">

<span class="input-group-addon" id="basic-addon1">@</span>

<input type="text" class="form-control" placeholder="Username" aria-describedby="basic-addon1">

</div>

<div class="input-group">

<input type="text" class="form-control" placeholder="Recipient's username" aria-describedby="basic-addon2">

<span class="input-group-addon" id="basic-addon2">@example.com</span>

</div>

<div class="input-group">

<span class="input-group-addon">$</span>

<input type="text" class="form-control" aria-label="Amount (to the nearest dollar)">

<span class="input-group-addon">.00</span>

</div>

<label for="basic-url">Your vanity URL</label>

<div class="input-group">

<span class="input-group-addon" id="basic-addon3">https://example.com/users/</span>

<input type="text" class="form-control" id="basic-url" aria-describedby="basic-addon3">

</div>

Пример группировки форм ввода представлен на рисунке 4.9.



Рис. 4.9. Пример группировки формы ввода.

**Панель навигации**

Доступная в Bootstrap панель навигации имеет общую разметку, начиная с базового класса .nav. Осуществляется обмен модификаторами классов для переключения между каждым стилем. Для создания вкладок в панели навигации используйте класс .nav-tabs. Заметьте, что класс .nav-tabs требует базового класса .nav:

<ul class="nav nav-tabs">

<li role="presentation" class="active"><a href="#">Home</a></li>

<li role="presentation"><a href="#">Profile</a></li>

<li role="presentation"><a href="#">Messages</a></li>

</ul>

Пример панели навигации представлен на рисунке 4.10.

8

Рис. 4.10. Пример группы формы ввода.

**Формы в навигационой панели**

Разместите поля внутри формы, содержащей класс .navbar-form, для правильного вертикального выравнивания и обеспечения функцией свертывания при малых размерах окнах. Используйте параметры выравнивания чтобы определить, где будет находиться содержимое в навигационной панели:

<form class="navbar-form navbar-left" role="search">

<div class="form-group">

<input type="text" class="form-control" placeholder="Search">

</div>

<button type="submit" class="btn btn-default">Submit</button>

</form>

Данный код на Bootstrap будет выглядеть следующим образом:



Рис. 4.11. Пример формы в навигационной панели.

**Jumbotron**

Jumbotron – это легкий, гибкий компонент, который можно расширять на весь экран, чтобы продемонстрировать ключевой контент вашего сайта. Рассмотрим пример Jumbotron:

<div class="jumbotron">

<h1>Hello, world!</h1>

<p>...</p>

<p><a class="btn btn-primary btn-lg" href="#" role="button">Learn more</a></p>

</div>

Пример использования Jumbotron представлен на рисунке 4.12.

Рис. 4.12. Пример Jumbotron.

### Плагины

В Bootstrap плагины могут подключаться отдельно (используя отдельные файлы Bootstrap\*.js), или все за один раз (используя bootstrap.js или минимизированый bootstrap.min.js).

**Атрибуты данных**

Вы можете использовать все Bootstrap плагины исключительно на основе предоставленного API без необходимости писать код на JavaScript. Тем не менее, в некоторых ситуациях может быть необходимым отключения этой функции. Таким образом, Bootstrap также предоставляет возможность отключить API-атрибуты, через отвязку всех событий в пространстве имен документа, при помощи data-api. Это выглядит следующим образом:

$(document).off('.data-api')

Также предоставляется возможность отключить определенный плагин, просто добавив имя плагина рядом с пространством имен data-api, как в примере:

$(document).off('.alert.data-api')

**Программный API**

Функциональность Bootstrap позволяет применять все плагины с использованием JavaScript API. Вызовы методов публичного API могут последовательно связываться:

$('.btn.danger').button('toggle').addClass('fat')

Все методы принимают в качестве необязательного параметра или объект, или строку, который предназначен для настройки функциональности активного элемента:

$('#myModal').modal() // инициализируется по умолчанию

$('#myModal').modal({keyboard: false})// инициализируется без клавиатуры

$('#myModal').modal('show') // инициализирует и запускает сразу

**Модальные окна**

Модальное окно является гибким диалоговым окном с минимальной функциональностью и необходимыми настройками по умолчанию. Открытие нескольких модальных окон не поддерживается.

Рассмотрим несколько примеров модальных окон.

На рисунке 4.13 представлен пример модального окна с заголовком, основным содержанием и перечнем действий:

<div class="modal fade" tabindex="-1" role="dialog">

<div class="modal-dialog" role="document">

<div class="modal-content">

<div class="modal-header">

<button type="button" class="close" data-dismiss="modal" aria-label="Close"><span aria-hidden="true">&times;</span></button>

<h4 class="modal-title">Modal title</h4>

</div>

<div class="modal-body">

<p>One fine body&hellip;</p>

</div>

<div class="modal-footer">

<button type="button" class="btn btn-default" data-dismiss="modal">Close</button>

<button type="button" class="btn btn-primary">Save changes</button>

</div>

</div>

</div>

</div>

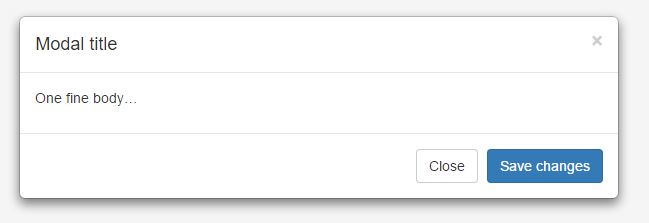


Рис. 4.13. Пример простого модального окна

Выпадающее меню

Выпадающее меню может быть размещено в любом месте, в том числе в навигационной панели, закладке и навигационных кнопках. На рисунке 4.14 и 4.15 представлены различные варианты выпадающего меню.

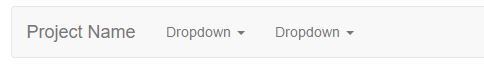


Рис. 4.14. Меню внутри навигационной панели.

14

Рис. 4.15. Меню внутри навигационных кнопок.

Подобные выпадающие меню могут быть реализованы следующим кодом:

<div class="dropdown">

<button id="dLabel" type="button" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">

Dropdown trigger

<span class="caret"></span>

</button>

<ul class="dropdown-menu" aria-labelledby="dLabel">

...

</ul>

</div>

Всплывающие подсказки

Для выведения различной информации, могут использовать различные всплывающие подсказки (см. рисунок 4.16):

<button type="button" class="btn btn-lg btn-danger" data-toggle="popover" title="Popover title" data-content="And here's some amazing content. It's very engaging. Right?">

Click to toggle popover

</button>

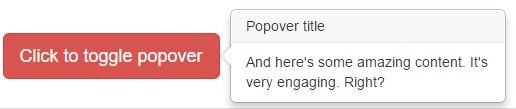


Рис. 4.16. Пример всплывающей подсказки.

Таким образом, в Bootstrap содержится большое количество инструментов, облегчающих жизнь разработчика:

1. Сетка − адаптивная 12-колоночная сетка с фиксированными размерами колонок.
2. Шаблон − может быть резиновым или фиксированным.
3. Типографика − можно оформить код, цитаты, абзацы, заголовки, заголовки со вторичным текстом, подзаголовки, выравнивание текста, аббревиатуры и т.п. Оформление уже прописано в CSS-классах, вам достаточно подключить стили и сделать верную разметку документа классами.
4. Медиа − позволяет красиво оформлять картинки и видео.
5. Таблицы − можно оформить таблицу, в том числе добавив возможность сортировки.
6. Навигация − для создания навигационного меню на сайте. Bootstrap упрощает ее, предлагая закладки, постраничную навигацию, "хлебные крошки", вкладки, боковое и главное меню и пр.
7. Формы − можно создавать различные формы: в одну или несколько строк, с подсказками и валидацией полей, с радио- или чекбоксами, лейблами и выпадающими кнопками. При этом можно подсвечивать неправильно заполненную часть формы, давая предупреждение пользователю при возникновении ошибки в заполнении.
8. Алерты − для оформления диалоговых окон. В основном это всплывающие окна или подсказки − ошибка (danger), предупреждение (warning), успех (success), подсказка (info).
9. Кнопки − в т.ч. «выпадающие» кнопки.
10. Прогресс-бары.
11. Шрифт из иконок − прекрасная возможность добавить красивых элементов в [оформлении сайта](http://blogwork.ru/oformlenie-sajta/). Правда с ограничениями: у исходной копии иконки может быть только один цвет.

Bootstrap позволяет не создавать свои CSS-конструкции, а пользоваться готовым решением и соредоточиться на функциональности web-приложения.

## **Вопросы для самоконтроля и задачи**

1. Кто ввел термин «адаптивный web-дизайн»?
2. Чем различаются резиновый и адаптивный дизайны?
3. Для чего нужна библиотека Bootstrap и каковы ее основные преимущества?
4. Как подключить библиотеку Bootstrap?
5. Что из себя представляют основные инструменты Bootstrap?
6. Что делает код <meta name="viewport" content="width=device-width">?
7. Какие текстовые редакторы для создания адаптивных web-конструкций вы знаете?

Список литературы

1. Гоше Х. Д. HTML5 для профессионалов [Текст] / Х. Д. Гоше ; пер. с англ. Е. Шикаревой, В. Крупника. - 2-е изд. - СПб. : Питер, 2015. - 559 с.
2. Дронов В. А.    HTML 5, CSS 3 и Web 2.0. Разработка современных Web-сайтов [Текст] / В. А. Дронов. - СПб. : БХВ-Петербург, 2014. - 414 с.
3. Дунаев В. В.    Самоучитель основы Web-дизайна [Текст] / В. В. Дунаев. -   
   2-е изд., перераб. и доп. - СПб. : БХВ -Петербург, 2013. - 479 с.
4. Ллойд Й.    Создай свой веб-сайт с помощью HTML и CSS [Текст] / Й. Ллойд ; пер. с англ. О. Потаповой. - СПб. : Питер, 2013. - 412 с.
5. Макфарланд Д. Большая книга CSS3. 3-е изд. [Текст] : пер. с англ. / Д. Макфарланд - СПб.: Питер, 2014. - 608 с.: ил
6. Муссиано Ч.    HTML и XHTML. Подробное руководство [Текст] : пер. с англ. / Ч. Муссиано, Б. Кеннеди ; пер. с англ. С. Иноземцева. - 6-е изд. - СПб. : Символ-Плюс, 2008.
7. Никсон Р.    Создаем динамические веб-сайты с помощью PHP, MySQL, JavaScript, CSS HTML5 [Текст] / Р. Никсон ; пер. с англ. Н. Вильчинского. - 3-е изд. - СПб. : Питер, 2015.
8. Прохоренок Н. А.    HTML, JavaScript, PHP и MySQL. Джентльменский набор Web-мастера [Текст] / Н. А. Прохоренок, В. А. Дронов. - 4-е изд., перераб. и доп. - СПб. : БХВ-Петербург, 2015. - 747 с.
9. Роббинс Д.    HTML5, CSS3 и JavaScript. Исчерпывающее руководство [Текст] / Д. Роббинс ; пер. с англ. М. А. Райтмана. - М. : ЭКСМО, 2014. – 528 с.
10. Робсон Э. Изучаем HTML, XHTML и CSS [Текст] / Э. Робсон, Э. Фримен ; пер. с англ. В. Черник. - 2-е изд. - СПб. : Питер, 2015. - 720 с.
11. Ташков П. А.    Веб-мастеринг : HTML, CSS, JavaScript, PHP, CMS, AJAX, раскрутка [Текст] / П. А. Ташков. - СПб. : ПИТЕР, 2010. - 512 с.
12. Хоган Б. HTML5 и CSS3. Веб-разработка по стандартам нового поколения [Текст] / Брайан Хоган СПб. : Питер, 2011. – 524 с.
13. Хольцшлаг М.    Языки HTML и CSS: для создания Web-сайтов [Текст] : [учеб. пособие] / М. Хольцшлаг ; пер. с англ. А. Климович. - М. : ТРИУМФ, 2006. - 303 с.
14. Чебыкин Р.И. Самоучитель HTML и CSS. Современные технологии [Текст] / Р.И. Чебыкин - СПб.: БХВ-Петербург, 2008. - 624 с.
15. Фельке-Моррис Большая книга веб-дизайна [Текст] / Терри Фельке-Моррис. - М. : ЭКСМО, 2012. - 606 с.
16. Фрэйн Б. HTML5 и CSS3. Разработка сайтов для любых браузеров и устройств. 2-е изд. - СПб.: Питер, 2017. - 272 с.
17. Электронный ресурс. URL: http://htmlbook.ru (дата обращения : 20.04.2019).

Оглавление

[ПРЕДИСЛОВИЕ 3](#_Toc25799067)

[Глава 1. Язык программирования JavaScript 4](#_Toc25799068)

[1.1 Введение в JavaScript 4](#_Toc25799069)

[1.2 Основы языка JavaScript 5](#_Toc25799070)

[1.3 JavaScript в браузере 38](#_Toc25799071)

[1.4 Дерево DOM 39](#_Toc25799072)

[1.5 События 70](#_Toc25799073)

[1.6 Вопросы для самоконтроля и задачи 78](#_Toc25799074)

[Глава 2. Библиотека jQuery 80](#_Toc25799075)

[2.1 Почему jQuery? 80](#_Toc25799077)

[2.2 Подключение jQuery 80](#_Toc25799078)

[2.3 Выбор элементов в jQuery 82](#_Toc25799079)

[2.4 Работа с элементами 88](#_Toc25799080)

[2.5 Перемещение элементов DOM 94](#_Toc25799081)

[2.6 Обход дерева DOM 97](#_Toc25799082)

[2.7 События 99](#_Toc25799083)

[2.8 Эффекты 102](#_Toc25799084)

[2.9 AJAX 104](#_Toc25799085)

[2.10 Компоненты 106](#_Toc25799086)

[Глава 3. Стандарт HTML 5 107](#_Toc25799087)

[3.1 Что такое HTML 5? 107](#_Toc25799089)

[3.2 Новые элементы 108](#_Toc25799090)

[3.3 Глобальные атрибуты 113](#_Toc25799091)

[3.4 Преобразование xhtml в HTML 5 115](#_Toc25799092)

[3.5 API интерфейсы 129](#_Toc25799093)

[3.6 Аудио и видео 130](#_Toc25799094)

[3.7 Canvas 133](#_Toc25799095)

[3.8 Web Storage и File API 137](#_Toc25799096)

[3.9 Средства взаимодействия с сервером 141](#_Toc25799097)

[3.10 Другие API-интерфейсы 143](#_Toc25799098)

[3.11 Вопросы для самоконтроля и задачи 150](#_Toc25799099)

[Глава 4. Адаптивная верстка 152](#_Toc25799100)

[4.1 Основы адаптивной верстки 152](#_Toc25799102)

[4.2 Особенности адаптивного дизайна 154](#_Toc25799103)

[4.3 Медиазапросы 159](#_Toc25799104)

[4.4 Библиотека Bootstrap 167](#_Toc25799105)

[4.5 Вопросы для самоконтроля и задачи 181](#_Toc25799106)

[Список литературы 182](#_Toc25799107)